This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problems Mailbox.

This Page Blank (uspto)

(19) Europäisches Patentamt
Europ an Patent Office
Office uropéen des brevets

(11) Publication number: **0 592 146 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: **93307757.0**

(51) Int. Cl.[5]: **H04N 1/46**

(22) Date of filing: **30.09.93**

(30) Priority: **28.10.92 US 967055**
05.10.92 US 956300

(43) Date of publication of application:
13.04.94 Bulletin 94/15

(84) Designated Contracting States:
DE FR GB IT

(71) Applicant: **CANON INFORMATION SYSTEMS, INC.**
3188 Pullman Street
Costa Mesa, CA 92626 (US)

(72) Inventor: **Ruetz, Brigitte**
255 Lassen drive
San Bruno, California 94066 (US)

(74) Representative: **Beresford, Keith Denis Lewis et al**
BERESFORD & Co. 2-5 Warwick Court High Holborn
London WC1R 5DJ (GB)

(54) Color reproduction method and apparatus.

(57) Method and apparatus for color printing according to a printer table having high color smoothness for out-of-gamut colors. A color printer gamut edge is first determined and color primary values for colors within the printer gamut are calculated and inserted into the printer table. For transition colors, namely those colors outside the printer gamut edge but still within the printer table, color primary values are calculated by selecting a color from the printer gamut edge at a point which lies at a constant angle from the transition color in question. The same constant angle is used for each and every one of the transition colors. A border table is provided for colors outside of the printer table; the color primary values for the border table are selected in the same manner as for the transition colors. According to the invention, it is possible to provide smooth color transitions for out-of-gamut colors, and in particular it is possible to provide monotonically increasing lightness in both the printer table and the border table and thereby avoid undesirable bands of darker colors within lighter colors.

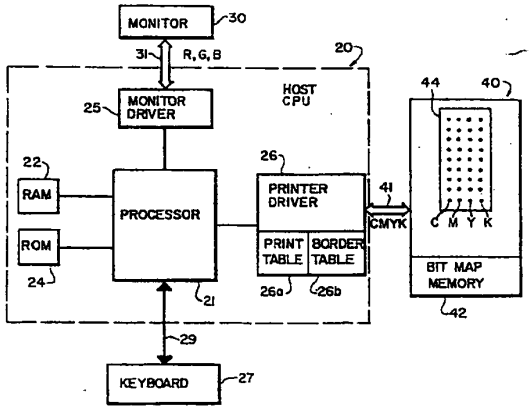

FIG. 3

EP 0 592 146 A2

RELATED APPLICATIONS

This application is related to a copending European patent application (agent's ref. 2271730), entitled "Method And Apparatus For High Fidelity Color Reproduction", the contents of which are incorporated by reference as if set forth here in full.

BACKGROUND OF THE INVENTION

This application includes an appendix of computer program listings.

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the document or the patent disclosure, as it appears in the Patent Office patent file or records, but otherwise reserves all copyright rights whatsoever.

Field Of The Invention

The present invention pertains to a method and apparatus for building and using look-up tables which determine the colors that a color printer prints in response to requests to print specific colors. The specific colors requested to be printed may include colors that are not printable by the printer. For those colors in particular, the colors in the look-up tables vary smoothly so that there are not large, discontinuous jumps in printed color for only a small change in requested color. In addition, lightness for those colors changes monotonically, that is, for requests to print increasingly lighter shade of colors the actual colors printed are also increasingly lighter without any undesirable dips towards darker colors.

Description Of The Related Art

Recently, as the availability of color monitors and color printers has increased, it is more and more commonplace for a computer user to view a full color image on a color monitor and then to request a full color printout of that image on a color printer.

However, color printers and color monitors form color images differently. Specifically, a color monitor is a light emitting device; colors are formed on color monitors by adding light from three color primaries, generally red, green and blue. Printed images, on the other hand, simply reflect ambient light; colors are perceived by the way ambient light is affected by three subtractive primaries, generally cyan, magenta and yellow (and sometimes black).

These processes are fundamentally different. As a result, the range of colors displayable on the monitor is different from the range of colors printable by a printer. Figure 1 is the CIE 1931 chromaticity diagram showing the range (or "gamut") of colors displayable by a monitor (area "A") and the range (or "gamut") of colors printable by a printer (area "B"). As seen in Figure 1, the range of colors displayable on a monitor is generally greater than the range of colors printable by a printer. This is because a monitor is a light emitting device and is able to display colors with greater saturation. There are, however, some low saturation areas such as at area 10 where a printed image, which is light-subtractive, has greater color range than a monitor.

Because of the difference between the ranges of printable and displayable colors, it has not heretofore been possible to print color images which are perceived as faithful reproductions of displayed color images. Specifically, it is simply not possible to print a color in areas like out-of-gamut area 11 which are outside the range "B" of printable colors. Accordingly, even though those colors may be seen on color monitors, they cannot be printed on a color printer.

In U.S. Patent 4,941,038, out-of-gamut colors were adjusted to colors on the exterior of the printer gamut which had the shortest vector distance to the out-of-gamut colors and which preserved the chroma and hue of the out-of-gamut colors. But because each out-of-gamut color was adjusted independently, out-of-gamut colors are printed with poor color smoothness, where small changes in commanded color can result in large changes in printed color. In particular circumstances, poor color smoothness manifests itself as non-monotonic changes in luminance whereby the lightness of out-of-gamut colors does not increase smoothly and monotonically from dark to light but rather dips occasionally from light to dark. This results in a situation where colors which should merge smoothly and monotonically from dark to light in fact show undesirable bands of darkness.

Figure 2 shows an example of non-smooth color transitions in the form of non-monotonic changes in lightness. Figure 2 shows a cross-section along an arbitrary hue plane in the a* and b* axes of CIELAB space and projected alone the L* lightness axis. In Figure 2, 24 designates the edge of the color printer gamut. Within edge 24, colors are printable and outside edge 24 the colors are out-of-gamut, unprintable colors. 26 designates

a border within which out-of-gamut colors are mapped into printable colors on the edge 24 of the printer gamut. Each of rows 27 represents a table which gives CMY values to be printed in response to a command to print the color in the corresponding color space. For purposes of illustration, Figure 2 only shows the luminance value L* which results when the CMY values are printed.

Ordinarily, it is expected for luminance to increase smoothly in directions parallel to th L* axis. Thus, for column 28 which is parallel to the L* axis, the luminance L* value increases smoothly from L* = 50 to 56.

For some areas, however, like column 29, the luminance starts to rise from L* = 50 to 52, but then dips back to L* = 51 before rising to the final level of L* = 52. Thus, the change in luminance in the resulting printed colors is non-monotonic showing undesirable dips or bands of darkness.

Non-monotonic changes in lightness is only one example of non-smooth color transitions for out-of-gamut colors.

SUMMARY OF THE INVENTION

It is an object of the present invention to address the foregoing difficulties and to provide printer tables whose colors vary smoothly in the out-of-gamut regions, and in particular, whose colors in the out-of-gamut region exhibit monotonic increases in lightness.

According to this aspect of the invention, transition colors are selected from colors on the edge of the printer gamut which are at a constant angle from the transition color without regard for change in lightness. Because each of the transition colors are at a constant angle from the edge of the printer gamut, the situation shown in Figure 2 cannot arise. Accordingly, out-of-gamut colors vary smoothly and exhibit monotonic increase in lightness for increasingly lighter out-of-gamut colors.

This brief summary has been provided so that the nature of the invention may be understood quickly. A more complete understanding of the invention can be obtained by reference to the following detailed description of the preferred embodiment thereof in connection with the drawings which together form a complete part of the specification.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a chromaticity diagram showing how the gamut of colors printable on a printer is related to the gamut of colors displayable on a monitor.

Figure 2 illustrates printer tables in a cross-section of CIELAB color space.

Figure 3 is a block diagram of a printing apparatus according to the invention.

Figure 4 is a flow diagram showing how a printer driver in the Figure 3 apparatus selects CMYK values for a color printer.

Figure 5 is a flow diagram for describing how the printer table and the border table are constructed.

Figure 6 illustrates a typical division of CIELAB space into a printer table.

Figure 7 shows how unconnected regions are removed from the printer table.

Figure 8 shows radially concave regions in the printer gamut and Figure 9 illustrates how to remove concavities by making the printer table radially convex.

Figure 10 shows how CMY values are selected for each cell in the transition region of the printer tables.

Figure 11 illustrates the arrangement of border tables.

Figure 12 shows the relation between the printer table and the border table in CIELAB space for an arbitrary luminance value L*.

Figure 13 shows the printer gamut at an arbitrary luminance value L* in the a* and b* axes.

Figure 14 shows how hue angles are warped for printer tables and Figure 15 shows how hue angles are warped for border tables.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Figure 3 is a block diagram of a printing apparatus according to the invention.

As shown in Figure 3, the printing apparatus includes a host CPU 20, a color monitor 30 and a color printer 40. Host CPU 20 includes a processor 21 such as an 80286 microprocessor, a random access memory ("RAM") 22 which provides working storage area to processor 21, a read only memory ("ROM") 24 which provides static storage for processor 21, monitor driver 25 and a printer driver 26. Host CPU 20 is accessed by an operator via keyboard 27 which is connected through an interface 29 to processor 21. Using the keyboard, an operator can cause processor 21 to execute stored program instructions which cause color images to be displayed on monitor 30 and which cause corresponding color images to be printed on color printer 40.

Other peripheral devices, such as disk drives, tape drives, color video interfaces, color scanner interfaces, etc., may be provided for host CPU 20 but those other devices are not shown in th interest of simplicity. In cooperation with the stored program instructions executed by processor 21, such devices permit, for example, a color image to be scanned into RAM 22 and displayed on monitor 30, the colors in the image to be manipulated, and the resulting image to be printed on printer 40.

In accordance with stored program instructions, processor 21 derives a color image for display on monitor 30. Processor 21 provides the color image to monitor driver 25 which in turn derives RGB values for each pixel in monitor 30. The RGB values are provided via interface 31 to the monitor 30 where those values are displayed.

Upon request, processor 21 also feeds a color image to printer driver 26 for printing by color printer 40. Printer driver 26 derives CMY values for each pixel of the color image based on the color values provided from processor 21. The CMY values are determined in accordance with either a printer table 26a or a border table 26b. The printer table 26a is a table which provides CMY values for all colors that are printable by printer 40. The border table 26b is a table which provides suitable CMY values for colors that are not printable by printer 40. The printer table may also include CMY values for some unprintable colors so as to smooth the transition from printable to unprintable colors. In addition, a black (hereinafter "K") value may also be derived. The CMYK values are fed via interface 41 to printer 40 where they are stored in bit map memory 42 within printer 40. The bit map memory 42 may store a full bit map image of the printed image or it may store only a band or partial bit map image. When sufficient color data is stored in bit map memory 42, a color printer head 44 reciprocates across a platen adjacent to a sheet of paper. In the present embodiment, print head 44 includes 32 ink jet nozzles arranged in a four column by eight row pattern. The nozzles in the first column all eject droplets of cyan ink; the nozzles in the second column all eject droplets of magenta ink; the nozzles in the third column all eject droplets of yellow ink; and the nozzles in the fourth column all eject droplets of black ink. The nozzles are controlled independently in accordance with the color data in bit map memory 42 such that in one reciprocation of print head 44 across the platen, eight rows of pixels are printed.

Figure 4 is a flow diagram showing how printer driver 26 selects CMYK values from the color data provided by processor 21. In step S401, printer driver 26 receives RGB values for a location (x,y) in bit map memory 42. In step S402, printer driver 26 derives device independent color coordinates from the RGB value. Preferably, the device independent coordinates are CIELAB coordinates. This is because the coordinates in CIELAB space are perceptually uniform such that equal-sized intervals anywhere in CIELAB space correspond to equal-sized changes in perceived color. Moreover, since CIELAB space can be viewed in cylindrical coordinates in terms of hue, saturation and luminance, it is an intuitive space which is amenable to defining gamut maps.

In step S403, the luminance coordinate is compressed at the extremes of the L* axis in CIELAB space. Compression step S403 may be performed explicitly by mathematical manipulation of the L* value from step S402 or implicitly by storing modified CMY values in the printer table and the border table. If performed implicitly, which is preferable in some instances, then both the printer table and the border table store pre-compressed values. More particularly, the printer table and border table can be arranged so that the values stored at, for example, luminance L* = 99 actually correspond to a luminance of L* = 94. Likewise, values-stored at, say, luminance L* = 7 actually correspond to a luminance of L* = 26. Values in the center of the luminance range, from for example L* = 38 through 90, remain unmodified. This arrangement results in luminance compression without the need for explicit compression.

While compression step S403 is optional, it is nevertheless preferable to perform since it ensures that colors at extreme values of luminance are printed with perceptible changes in luminance. More particularly, because monitor 30 displays colors with light emitting elements, it can display colors with higher values of luminance than those achievable by printer 40, whose highest value of luminance is limited by the whiteness of the paper upon which the color image is formed. Further still, since monitor 30 can completely turn off its light emitting elements, it can display colors with lower values of luminance than those printable by printer 40, since even black ink reflects some ambient light. Accordingly, to ensure that some color is printed, even at the highest and lowest luminance values, it is preferable to compress the luminance values determined in step S402 into a range that is printable by printer 40.

In step S404, the L*, a* and b* coordinates derived in steps S402 and S403 are inspected to determine whether they fall within the range covered by printer table 26a. If the L*, a*, b* coordinates are within the range covered by printer table 26a, th n flow advances to step S405 which looks up the corresponding CMY values in printer table 26a at location L*, a*, b* (actually, the nearest location since only discrete values of L*, a* and b* are stored). On the other hand, if the L*, a*, b* coordinates are not within printer table 26a, then flow advances to step S406 in which the hue angle θ is d rived from the a* and b* values according to the following formula:

$$\theta = \arctan(b^*/a^*)$$

Flow then advances to step S407 which looks up corresponding CMY values in border table 26b at the nearest location which corresponds to the luminance L* and the hue angle derived in st p S406.

In either event, flow then advances to step S408 in which the CMY valu s are stored in bit map memory 42 at location (x,y). If desired, the CMY values may be modified before storage, for example, by interpolation, so as to accommodate the difference between the actual L*, a*, b* values stored in the tables and the desired values calculated above.

In step S409, printer driver 26 determines whether the bit map memory has been completed. If the bit map memory has not been completed, then flow returns to step S401 in which the next RGB value is received for the next location (x,y) in bit map memory. On the other hand, if the bit map memory has been completed, or if a sufficient area of the bit map memory has been completed (such as an eight row long band corresponding to the eight rows of ink jet nozzles in head 44), then flow advances to step S410 where gamma correction is performed. Gamma correction adjusts the CMY values in bit map memory so as to achieve a uniform distribution of luminance. In step S411, undercolor removal is performed so as to derive the black value for location (x,y) in bit map memory. Undercolor removal in the present embodiment may be performed by the simple expedient of selecting the minimum value of CMY and assigning that value to the black value. Then, each of the CMY values is adjusted by subtracting the black value from it.

The order of steps S410 and S411 is not critical and those steps may be switched, for example to accommodate a particular color printing technique such as continuous tone, dither techniques or error diffusion.

In step S412, color printing is initiated using the resulting CMYK values.

Figure 5 is a flow diagram for describing how the printer table 26a and the border table 26b are constructed. The flow procedures shown in Figure 5 need only be performed once for each printer, or once whenever it is desired to re-calibrate each printer. More preferably, the flow procedures of Figure 5 are performed only once for a class of printers, such as printers of the same model number, and are provided in software to an operator as part of a factory calibration of the printer.

In step S501, the gamut or range of colors printable by printer 40 is measured. Preferably, this is achieved by printing a very large subset, or a complete set, of all colors printable by printer 40. For example, in the printer used in the present embodiment, each of the CMY and K values may be printed in 65 gradations ranging numerically from 0 through 64. A subset of about one quarter of those values, for each color, are printed. Thus, for example, 17 C values are printed, namely numerical values 0, 4, 8, 12, ... 64, and 17 M values are printed, and 17 Y values are printed. All possible combinations of those 17 CMY values are printed, yielding 17 x 17 x 17 = 4,913 color patches.

In addition to the foregoing hued colors, all possible gray values, in this case 48 additional gray values over the 17 already printed, are also printed.

With the foregoing sampling of the printer gamut, it will be seen that pure gray colors are printed together with hued colors. Whatever method of sampling is used, this property of pure gray printing should be preserved since proper gray reproduction is a desirable property in color reproduction.

The color of each of the 4,913 color patches and 48 additional gray patches is measured in a device independent color space such as the aforementioned CIELAB color space. Thus, at the end of step S501, for each of the 4,913 + 48 = 4,961 unique CMY color combinations, L*, a* and b* coordinates are measured thereby defining the printer gamut.

Step S502 derives mathematically smooth functions which map the CIELAB coordinates into CMY coordinates. In the present embodiment, a cubic least squares fit from CIELAB space into CMY space was chosen. That is, using well-known least squares fitting techniques, coefficients c_0 through c_{19}, m_0 through m_{19}, and y_0 through y_{19} were derived to give the best fit, in the least squared sense, to the gamut measured in step S501:

$$\begin{aligned} C = {} & c_0 + c_1L^* + c_2a^* + c_3b^* + c_4L^{*2} + c_5a^{*2} + c_6b^{*2} \\ & + c_7L^*a^* + c_8L^*b^* + c_9a^*b^* + c_{10}L^{*3} + c_{11}a^{*3} + \\ & c_{12}b^{*3} + c_{13}L^{*2}a^* + c_{14}L^*a^{*2} + c_{15}L^{*2}b^* + c_{16}L^*b^{*2} \\ & + c_{17}a^{*2}b^* + c_{18}a^*b^{*2} + c_{19}L^*a^*b^* \end{aligned} \quad (1)$$

$$M = m_0 + m_1L^* + m_2a^* + m_3b^* + m_4L^{*2} + m_5a^{*2} + m_6b^{*2} + m_7L^*a^* + m_8L^*b^* + m_9a^*b^* + m_{10}L^{*3} + m_{11}a^{*3} + m_{12}b^{*3} + m_{13}L^{*2}a^* + m_{14}L^*a^{*2} + m_{15}L^{*2}b^* + m_{16}L^*b^{*2} + m_{17}a^{*2}b^* + m_{18}a^*b^{*2} + m_{19}L^*a^*b^* \quad (2)$$

$$Y = y_0 + y_1L^* + y_2a^* + y_3b^* + y_4L^{*2} + y_5a^{*2} + y_6b^{*2} + y_7L^*a^* + y_8L^*b^* + y_9a^*b^* + y_{10}L^{*3} + y_{11}a^{*3} + y_{12}b^{*3} + y_{13}L^{*2}a^* + y_{14}L^*a^{*2} + y_{15}L^{*2}b^* + y_{16}L^*b^{*2} + y_{17}a^{*2}b^* + y_{18}a^*b^{*2} + y_{19}L^*a^*b^* \quad (3)$$

In step S502, any mathematical function which fits the measurements taken in step S501 from the device independent coordinate space to CMY coordinate space may be used. Preferably, however, the mapping function includes smoothing so as to eliminate measurement irregularities that may have been encountered in step S501.

It may, in addition, be preferable to weight some of the points measured in step S501 prior to deriving mapping in step S502. For example, proper skin tone color reproduction is an important property of color printers. Accordingly, it may be desirable, in some circumstances, to weight colors in the area of skin tone colors more heavily than other colors.

In step S503, the device independent space, namely CIELAB space, is divided into equally sized intervals, one of the intervals including the L* axis such as by being centered at the L* axis, thereby providing a blank printer table. The size of the printer table preferably includes both the printer gamut as well as the gamut of a typical color monitor. For example, referring to Figure 1, the printer table preferably includes the color area indicated generally at 12. The size of the intervals in the printer table should be made as small as possible giving due consideration to storage limitations for the printer table. Thus, for example, it has been found that fine luminance gradations are more important than fine hue and saturation gradations. It has been determined that dividing the luminance axis into intervals of $\Delta L^* = 1$ (luminance L* ranges from 0 through 100) provides adequate luminance gradation. On the other hand, such fine gradations are not ordinarily needed in hue, and therefore $\Delta a^* = \Delta b^* = 3$ has been found to yield adequate hue gradations (a* and b* range from about -100 through +100 near the center (L* = 50) of the luminance axis).

In addition to the foregoing considerations, it should also be observed that the printer gamut is not the same for each luminance value. Specifically, the gamut is relatively smaller at luminance extremes and relatively larger at the center of the luminance axis. Figure 6 illustrates a typical division of CIELAB space into a printer table, although all luminance and hue gradations have not been shown to simplify the presentation. At relatively low luminance values such as L* = 10, a relatively small rectangular grid in the a* and b* axes is adequate to map the printer gamut. Similarly, at relatively high luminance values, such as L* = 90, a relatively small rectangular grid in the a* and b* axes is also adequate to store the printer gamut. However, at intermediate luminance values, such as that at L* = 50, a relatively larger rectangular grid in the a* and b* axes is required to map the printer gamut.

As further shown in Figure 6, the rectangular grid at each luminance level includes the L* axis (in Figure 6 it is centered on the L* axis). That is, there is a cell in the rectangular grid that corresponds exactly to a* = b* = 0. That central point, namely a* = b* = 0, corresponds to a pure gray color which, as mentioned above, is desirably reproduced as a pure gray color for proper color reproduction.

In practice, it is also preferable to include more colors in the printer table than are in the printer gamut, and most preferably also to include the colors that are found in a typical monitor's gamut. This permits the printer table to include transition values which smooth the transition from colors at the edge of the printer gamut to colors in the border table and which preserve color differ ntiation in areas outside the printer gamut.

In step S504, C, M and Y values are inserted into the printer table around the L* axis using the mapping functions derived in step S502. For digital color printing as opposed to continuous tone printing, fractional C, M and Y values are truncated or rounded to integral values. The entire rectangular grid at each luminance level is not completely filled, but rather only those points known to be within the printer gamut. In addition, the points exactly on the L* axis, namely those points at a* = b* = 0, are also not mapped by the mapping functions. Rather,

CMY values for those points are inserted in st p S505 by determining printer grays measured in step S501 with corresponding L* values. This ensures that the smoothing introduced by the mapping functions does not introduce hue into pure gray values.

In steps S506 and S507, the CMY values in the printer table are adjusted for unprintable colors. Unprintable colors may arise because of artifacts introduced by the mapping function selected in step S502. For example, the mapping function used may give rise to false regions in the printer table, such as region 45 in Figure 7, which is not within the printer gamut 46. These artifacts are removed in step S506 by removing all regions that are not connected to the region around the L* axis.

Unprintable colors may also arise from the situation shown in Figure 8 in which 47 designates the edge of the printer gamut for an arbitrary luminance value L*. The printer gamut shown there is not radially convex because each and every radial line from the L* axis does not intersect edge 47 at one and only one point. In particular, radial line 48 intersects edge 47 at three points 49a, 49b and 49c. The region between points 49a and 49b is a radial concavity and can cause the generation of inappropriate CMY values in the printer table. Accordingly, in step S507, the values in the printer table are adjusted to make them radially convex.

Figure 9 illustrates this process. Figure 9 shows the rectangular grid in the a* and b* axis for an arbitrary luminance value L*. Cells 51 through 55 all include printable values within the printer gamut. However, cell 59 is a radially concave cell because the radial line at θ crosses two cells in the printer gamut (cells 53 and 56). Accordingly, a CMY value is assigned to cell 59 to make the table radially convex. The value is selected by preserving as much as possible the hue of the color (angle θ in Figure 9) and by selecting a saturation value which is nearest to that desired. Thus, in Figure 9, the value of cell 59 could be assigned a value near to the value of one of cells 51 through 55 in dependence upon which of cells 51 through 55 have the closer values of hue and saturation. In Figure 9, C = 1, M = 18 and Y = 14 has been selected.

Step S508 derives transition colors for each printer table, and Figure 10 shows how the CMY values for the transition colors are selected. In Figure 10, 60 is the edge of the printer gamut and 61 is the edge of the printer tables which, as mentioned above, corresponds roughly to the edge of a typical color monitor. For each transition color in the region between edges 60 and 61, the color on the edge 60 of the printer gamut which lies at a constant anglefrom the transition color is selected for the transition color. Any change in luminance is allowed, that is, the change in luminance is not limited to an arbitrary threshold. For example, for transition color 62 point 64 on the edge 60 of the printer gamut is selected because point 64 lies at an angle α from point 62. Likewise, for each point in the transition region, such as point 65, the corresponding point on the edge 60 the printer gamut which lies at a constant angle α is selected, here point 66. For transition colors above the maximum saturated point 67 of the printer gamut, the angle α extends downwardly; conversely, for transition colors below the maximum saturated point 67, such as color 69, the angle α extends upwardly. For transition colors in the wedge 70 subtended by the angle 2α from the maximum saturated point 67, the maximum saturated point 67 is selected. Thus, all colors in the wedge 70 are mapped to color 67.

To ensure maximal color smoothness for the transition colors between edges 60 and 61, the color saturation of the printer gamut edge 60 is preferably inspected before constant α angle extension to ensure that the color saturation increases monotonically from minimum saturation to the maximum saturation point and then decreases monotonically to minimum saturation. If non-monotonicity is found, then the color saturation at the printer gamut edge is adjusted before constant angle extension to remove the non-monotonicity.

An angle $\alpha = 15°$ has been found to yield satisfactory results by allowing a pleasing increase in color saturation without unreasonably large changes in lightness. Other values of α, for example between 10° and 20°, may also be used.

Step S509 derives the border table 26b. Whereas printer table 26a was arranged as a rectangular grid in the a*, b* axes for each luminance value, the border table is arranged in wheels with one wheel for each of the luminance values of the printer table. Thus, as shown in Figure 11, a wheel-shaped border table is provided for each of the luminance values for which a printer table exists, which provides one border table in correspondence with each of the printer tables. The border tables contain plural cells which are accessed based on hue angle θ calculated as a function of the a* and b* coordinates, as follows:

$$\theta = \arctan(b^*/a^*)$$

Figure 12 shows the correspondence of border tables and printer tables. Whereas printer table 26a is a rectangular grid for an arbitrary luminance value L*, border table 26b is arranged as a wheel centered at a* = b* = 0. Individual cells in the border table are accessed by the angle θ in the a* and b* axes which also corresponds to hue. Experimentally, it has been found that 1 degree increments, resulting in 360 cells in each border table, provides adequate gradation of hue, but this can be adjusted as described below in connection with Figure 13.

Step S510 selects CMY values for each cell in the border tables in the same manner as for selecting CMY values for transition colors in the printer table. Thus, border table colors are selected from the printer edge color

which lies at a constant angle α (which is the same α as used for the printer table) from the border table. As befor , α extends upwardly or downwardly depending on whether the border table color is below or above the maximum printer saturation point, and is limited to th maximum printer saturation point when the border color table falls in wedge 70.

In step S510, the border table values are inspected to ensure that the saturation of the colors in the border table changes smoothly. This situation is illustrated in Figure 13 which shows the printer gamut 70 at an arbitrary luminance value L* in the a* and b* axes. As described above, colors such as color 71 which lie outside of the printer gamut are mapped to border color 72 of the printer gamut by preserving hue angle. Particularly in areas like area 74 where the printable saturation changes rapidly, small changes in hue can cause rapid changes in saturation in the border table. For example, as the hue angle changes from θ_1 to θ_2, there are relatively large changes in saturation for only a small change in hue. Such a large change in saturation, when printed, looks unnatural.

To avoid this unnatural look, the border table is increased in size until there are. sufficiently fine increments to ensure that saturation changes smoothly. If the border table size is increased, then the calculations in step S509 are repeated to fill in CMY values for the new border table.

In step S511, the CMY values in the printer table are inspected and modified so as to ensure that they merge smoothly into the gray (L* axis). Specifically, at each discrete luminance level, the colors in the printer table that are close to the L* axis are redetermined so as to ensure that they converge smoothly to gray.

In step S512, the printer table is rectangularized. More specifically, until this step, CMY values have be n inserted into the printer table only in areas within the printer gamut 60 (steps S504 and S505) and in the transition region 61 between the printer gamut and the border table (step S509). In step S512, the remaining cells of the printer table such as cells like 69 in Figure 12 are filled out by calculating the hue angle for each blank cell that remains in the printer table and by inserting the border table color at that hue angle as illustrated at 68 in Figure 12.

In step S513, the hue angles in the printer table and in the border table are warped so as to compensate for the Abney effect. More specifically, the CMY values for out-of-gamut colors that are currently stored in the printer table and the border tables are all based on constant hue angle extension as described in connection with step S508. However, for highly saturated colors, constant hue angle extension back to the printer gamut edge can result in a change in perceived hue, in accordance with the Abney effect. For example, constant hue angle extension changes the hue of a highly saturated (but unprintable) purplish-blue color to a less-saturated purple color on the printer gamut edge.

To compensate for this effect, the hue angles in the printer table and in the border table are warped. More particularly, for both the printer table and the border table, the CMY values for one hue angle are transferred to another, different, hue angle so as to preserve the perceived hue of the printing color. This warping is illustrated in Figure 14 and 15 for the blue/purple region of color space which lies at a hue angle θ of between θ = 255° to θ = 333°.

Figure 14 shows a printer table 80 before warping and the same printer table 81 after warping. The printer tables illustrated in Figure 14 are for arbitrary printer tables in the a*, b* plane and for an arbitrary luminance value L*, and it should be understood that the warping shown in Figure 14 is carried out for each of the printer table for the L* values selected in step S503. As shown in Figure 14, blue region 82a is stretched into warped blue region 82b. More particularly, each of the CMY values in region 82a is stretched into corresponding warped regions in 82b. This ensures that the CMY values that are printed in response to a command to print a highly saturated out-of-gamut blue color yield a blue hued color rather than a purplish-blue color. For example, highly saturated out-of-gamut blue color 84, if printed according to the unwarped table 80 would yield a purplish-blue color while when printed according to warped table 81 yields a blue color.

Additional hue angle warpings map the CMY values from region 85a into region 85b and map CMY values from region 86a into region 86b. The precise mappings are given as follows:

For hue angles between 255° and 305°:

$$\text{warped_angle}[255 + \text{ang}] = 255 + .5 * \text{ang}$$

(where 0 < ang < 50°)

For hue angles between 305° and 309°:

$$\text{warped_angle}[305 + \text{ang}] = 280 + 1.25 * \text{ang}$$

(where 0 < ang < 4°)

For hue angles between 309° and 333°:

$$\text{warped_angle}[309 + \text{ang}] = 285 + 2 * \text{ang}$$

(where 0 < ang < 24°)

Thus, the unwarped region from 255° to 280° is warped by stretching into the region 255° to 305°, the unwarped region from 280° to 285° is warped by compressing into the region from 305° to 309°, and the unwarped

region from 285° to 333° is warped by compressing into the region from 309° to 333°. The warped regions remain continuous, however, and they have the same end points as the unwarped regions (here 255° and 333°).

Similar warpings are carried out in the red and cyan regions. For red, the warpings are:

For hue angles between 10° and 40°:

warped_angle [10 + ang] = 10 + 0.5 * ang
(where 0 < ang < 30°)

For hue angles between 40° and 53°:

warped_angle [40 + ang] = 25 + 1.25 * ang
(where 0 < ang < 12°)

For hue angles between 52° and 76°:

warped_angle [52 + ang] = 40 + 1.5 * ang
(where 0 < ang < 24°)

Thus, the unwarped region from 10° to 25° is warped by stretching into the region 10° to 40°, the unwarped region from 25° to 40° is warped by compressing into the region from 40° to 52°, and the unwarped region from 40° to 76° is warped by compressing into the region from 52° to 76°.

For cyan, the warpings are:

For hue angles between 170° and 195°:

warped_angle [170 + ang] = 170 + 2.0 * ang
(where 0 < ang < 25°)

For hue angles between 195° and 245°:

warped_angle [195 + ang] = 220 + 0.5 * ang
(where 0 < ang < 50°)

Thus, the unwarped region from 170° to 220° is warped by compressing into the region from 170° to 195°, and the unwarped region from 220° to 245° is warped by stretching into the region from 195° to 245°.

The same warping that is carried out on the printer tables is also carried out on the border tables as shown in Figure 15. Like before, this ensures that a request to print an out-of-gamut blue color, such as color 87, which would be printed as a purplish-blue hue if printed according to the unwarped border table, will result in a blue-hued color when printed according to the warped border table.

With the warping described above, color smoothness in the printer table and in the border table is preserved since both in-gamut and out-of-gamut colors are both warped. While it is possible to warp only out-of-gamut colors, this would result in an undesirable color discontinuity at the printer gamut edge. Moreover, even though the in-gamut colors are warped and hence are distorted, it has been confirmed experimentally that the amount of distortion is relatively insignificant because the hue angle rays are close together for colors in the printer gamut and are further apart for higher saturation out-of-gamut colors.

The above-described warping technique warps all color at the same hue angle equally regardless of the saturation of the colors. It is also possible to introduce a saturation-dependent warping factor, whereby more highly saturated colors are warped more than relatively less saturated colors.

In step S514, colors in the yellow region of the printer and border tables are adjusted so as to widen the yellow region. More particularly, as shown in Figure 2, pure yellow colors for the printer fall into a very narrow range of the printer gamut that has been found to be difficult for users to find (the range of monitor yellow colors is greater). Because the range of pure yellow printer colors is so narrow, a user typically obtains a greenish-yellow rather than the desired pure yellow color. Thus, in step S514, the yellow region is widened. Conveniently, yellow widening is obtained through hue angle warping as follows:

For hue angles between 87° and 91°:

warped_angle [87 + ang] = 87 + 1.25 * ang
(where 0 < ang < 4°)

For hue angles between 91° and 97°:

warped_angle [91 + ang] = 92
(where 0 < ang < 6°)

For hue angles between 97° and 112°:

warped_angle [97 + ang] = 92 + 0.5 * ang
(where 0 < ang < 15°)

For hue angles between 112° and 132°:

warped_angle [112 + ang] = 99.5 + 1.25 * ang
(where 0 < ang < 20°)

For hue angles between 132° and 147°:

warped_angle [132 + ang] = 124.5 + 1.5 * ang
(where 0 < ang < 15°)

A computer program for automatically performing the foregoing steps S501 through S514 has been developed and follows as an appendix to this description.

```
/*****************************************************************************

                MAKE-LUT (make_lut.c)

                Copyright (C) 1992 CANON INFORMATION SYSTEMS, INC.
                Copyright (C) 1992 CANON INC.

        This program uses the mathematical fit function from CIELAB to CMY,
        which was derived from the print patches measurements.
        It builds the "LUT" file, which lists the CMYs for all LAB intervals
        Together with the LUT it provides the "lab_borders" file, which
        shows the rectangular border of the LUT gamut at each lightness leve
        It also builds the "border_colors" file, which lists the CMYs for th
        out of gamut colors at all angles and lightness levels.
        All these files are binary.

                                                -- Brigitte Ruetz

*****************************************************************************/


#include "lut.h"

extern convert();
extern cmy cmy_vector;

static int startL, endL;
static int inside_gamut[L_STEPS][A_STEPS][B_STEPS];
static int transition[L_STEPS][A_STEPS][B_STEPS];
static int concave[L_STEPS][A_STEPS][B_STEPS];
static cmy LUT[L_STEPS][A_STEPS][B_STEPS];
static cmy border_color[L_STEPS][ANGLE_STEPS];
static double interval_l[L_STEPS], interval_a[A_STEPS], interval_b[B_STEPS];
static double saturation_l[ANGLE_STEPS];
static double warped_angle[ANGLE_STEPS];
static a_b rotated;
static cmy averaged;

int angle(a_index, b_index)      /* computes the polar coordinates angle */
int a_index, b_index;
{

  int original_a, original_b, rounded_angle;
  double fulltwo_angle, exact_angle;

  original_a = a_index - CENTER_A;
  original_b = b_index - CENTER_B;
  fulltwo_angle = atan2pi((double)original_b, (double)original_a);
  if (fulltwo_angle >= 0)
    exact_angle = (ANGLE_STEPS/2)*fulltwo_angle;
  else
    exact_angle = (ANGLE_STEPS/2)*(fulltwo_angle + 2.0);
  rounded_angle = (int)(exact_angle + 0.5);
  if (rounded_angle >= ANGLE_STEPS)
    rounded_angle = 0;
```

```
  return rounded_angle;
}

double radius(ab_vector)          /* computes the polar coordinates radius */
a_b ab_vector;
{
  return sqrt(ab_vector.a*ab_vector.a + ab_vector.b*ab_vector.b);
}

rotate(ang0, point0)              /* rotates a vector in the plane */
double ang0;
a_b point0;
{
  double co, si;

  ang0 = (2*ang0)/ANGLE_STEPS;
  co = cospi(ang0);
  si = sinpi(ang0);
  rotated.a = co*point0.a - si*point0.b;
  rotated.b = si*point0.a + co*point0.b;
}

average2(l0, a1, b1, a2, b2, round)   /* computes the average of two colors
int l0, a1, b1, a2, b2, round;
{
  int ang1, ang2;
  cmy vector1, vector2;

  if (a1 >= 0 && a1 < A_STEPS && b1 >= 0 && b1 < B_STEPS
    && (inside_gamut[l0][a1][b1] == 1 || transition[l0][a1][b1] == 1))
  {
    vector1.c = LUT[l0][a1][b1].c;
    vector1.m = LUT[l0][a1][b1].m;
    vector1.y = LUT[l0][a1][b1].y;
  }
  else
  {
    ang1 = angle(a1, b1);
    vector1.c = border_color[l0][ang1].c;
    vector1.m = border_color[l0][ang1].m;
    vector1.y = border_color[l0][ang1].y;
  }
  if (a2 >= 0 && a2 < A_STEPS && b2 >= 0 && b2 < B_STEPS
    && (inside_gamut[l0][a2][b2] == 1 || transition[l0][a2][b2] == 1))
  {
    vector2.c = LUT[l0][a2][b2].c;
    vector2.m = LUT[l0][a2][b2].m;
    vector2.y = LUT[l0][a2][b2].y;
  }
```

```
    else
    {
      ang2 = angle(a2, b2);
      vector2.c = border_color[10][ang2].c;
      vector2.m = border_color[10][ang2].m;
      vector2.y = border_color[10][ang2].y;
    }
    averaged.c = (unsigned char)((int)(0.5*((int)vector1.c
      + (int)vector2.c) + round*0.5));
    averaged.m = (unsigned char)((int)(0.5*((int)vector1.m
      + (int)vector2.m) + round*0.5));
    averaged.y = (unsigned char)((int)(0.5*((int)vector1.y
      + (int)vector2.y) + round*0.5));

}


average4(lk, ak, bk, round)
int lk, ak, bk, round;
{

  if (ak >= CENTER_A - 2 && ak <= CENTER_A + 2
    && bk >= CENTER_B - 2 && bk <= CENTER_B + 2)
  {
    averaged.c = LUT[lk][ak][bk].c;
    averaged.m = LUT[lk][ak][bk].m;
    averaged.y = LUT[lk][ak][bk].y;
  }
  else
  {
    if (concave[lk, ak - 1, bk] == 0 && concave[lk, ak + 1, bk] == 0)
      average2(lk, ak - 1, bk, ak + 1, bk, round);
    else
      average2(lk, ak, bk - 1, ak, bk + 1, round);
  }

}


void fit(ld, ad, bd, ci_ptr, mi_ptr, yi_ptr)     /* computes the CMY fits */
double ld, ad, bd;
int *ci_ptr, *mi_ptr, *yi_ptr;
{

  int ck, mk, yk;
  double a2,a3,ab,a2b,b2,ab2,b3,al,a2l,bl,abl,b2l,l2,al2,bl2,l3;
  double cd, md, yd;

  a2 = ad*ad;
  a3 = a2*ad;
  ab = ad*bd;
  a2b = a2*bd;
  b2 = bd*bd;
  ab2 = ad*b2;
  b3 = b2*bd;
  al = ad*ld;
  a2l = a2*ld;
  bl = bd*ld;
  abl = ad*bd*ld;
  b2l = b2*ld;
```

```
  l2 = ld*ld;
  al2 = ad*l2;
  bl2 = bd*l2;
  l3 = l2*ld;
  cd = FITC(ad,a2,a3,bd,ab,a2b,b2,ab2,b3,ld,al,a21,bl,abl,b21,l2,al2,bl2,l3)
  md = FITM(ad,a2,a3,bd,ab,a2b,b2,ab2,b3,ld,al,a21,bl,abl,b21,l2,al2,bl2,l3)
  yd = FITY(ad,a2,a3,bd,ab,a2b,b2,ab2,b3,ld,al,a21,bl,abl,b21,l2,al2,bl2,l3)
  if (cd >= 0)
    ck = (int)(cd + 0.5);
  else
    ck = (int)(cd - 0.5);
  if (md >= 0)
    mk = (int)(md + 0.5);
  else
    mk = (int)(md - 0.5);
  if (yd >= 0)
    yk = (int)(yd + 0.5);
  else
    yk = (int)(yd - 0.5);
  if (ck == 65)
    ck = 64;
  if (mk == 65)
    mk = 64;
  if (yk == 65)
    yk = 64;
  if (ck == -1)
    ck = 0;
  if (mk == -1)
    mk = 0;
  if (yk == -1)
    yk = 0;
  *ci_ptr = ck;
  *mi_ptr = mk;
  *yi_ptr = yk;

}


intersection(l0, ang0, point0,          /* intersects with printer gamut */
  updown, l_ptr, a_ptr, b_ptr)
int ang0, updown;
double l0, *l_ptr, *a_ptr, *b_ptr;
a_b point0;

{
  int lk, ak, bk, ah, bh, al, bl, ck, mk, yk, count;
  int reached, final_ah, final_bh, final_outside;
  double ls, l1, rad0, rad1, plane_ratio, warped_ang0;
  double a_width, b_width, final_l, final_a, final_b, l_step;
  a_b place, place1;

  ls = saturation_l[ang0];
  rad0 = radius(point0);
  al = (int)((point0.a - A_MIN)/DELTA_A);
  bl = (int)((point0.b - B_MIN)/DELTA_B);
  l_step = max2(0.0, 0.5*(l0 - dark));
  warped_ang0 = warped_angle[ang0];
  if ((warped_ang0 < ANGLE_STEPS/8 || warped_ang0 >= (7*ANGLE_STEPS)/8) ||
    (warped_ang0 >= (3*ANGLE_STEPS)/8 && warped_ang0 < (5*ANGLE_STEPS)/8))
```

```
{
  plane_ratio = tanpi((2.0*warped_ang0)/ANGLE_STEPS);
  if (al > CENTER_A)
  {
    for (ah = al; ah >= CENTER_A; ah--)
    {
      place.a = interval_a[ah];
      place.b = plane_ratio*place.a;
      rad1 = radius(place);
      if (rad1 > rad0)
        continue;
      l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
      rotate(ang0 - warped_ang0, place);
      ak = (int)((rotated.a - A_MIN)/DELTA_A);
      bk = (int)((rotated.b - B_MIN)/DELTA_B);
      for (lk = startL; lk <= endL; lk++)
      {
        if (interval_l[lk] + 0.5*(interval_l[lk + 1] - interval_l[lk]) > l
          break;
      }
      fit(l1, place.a, place.b, &ck, &mk, &yk);
      if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
          && yk >= 0 && yk <= 64
          && (inside_gamut[lk][ak][bk] == 1
            || (ak + 1 < A_STEPS
              && inside_gamut[lk][ak + 1][bk] == 1)
            || (ak + 1 < A_STEPS && bk + 1 < B_STEPS
              && inside_gamut[lk][ak + 1][bk + 1] == 1)
            || (bk + 1 < B_STEPS
              && inside_gamut[lk][ak][bk + 1] == 1)
            || (ak - 1 >= 0 && bk + 1 < B_STEPS
              && inside_gamut[lk][ak - 1][bk + 1] == 1)
            || (ak - 1 >= 0
              && inside_gamut[lk][ak - 1][bk] == 1)
            || (ak - 1 >= 0 && bk - 1 >= 0
              && inside_gamut[lk][ak - 1] [bk - 1] == 1)
            || (bk - 1 >= 0
              && inside_gamut[lk][ak][bk - 1] == 1)
            || (ak + 1 < A_STEPS && bk - 1 >= 0
              && inside_gamut[lk][ak + 1][bk - 1] == 1)))
        || inside_gamut[lk][ak][bk] == 1)
      {
        if (ck < 0 || ck > 64 || mk < 0 || mk > 64
          || yk < 0 || yk > 64)
          final_outside = 1;
        else
          final_outside = 0;
        final_l = l1;
        final_a = place.a;
        final_b = place.b;
        final_ah = ah;
        break;
      }
      else if (updown*(ls - l1) > 0)
      {
        final_outside = 0;
        final_l = l1;
        final_a = place.a;
        final_b = place.b;
        final_ah = ah;
```

```
            break;
          }
        }
      }
      else
      {
        for (ah = al; ah <= CENTER_A; ah++)
        {
          place.a = interval_a[ah];
          place.b = plane_ratio*place.a;
          rad1 = radius(place);
          if (rad1 > rad0)
            continue;
          l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
          rotate(ang0 - warped_ang0, place);
          ak = (int)((rotated.a - A_MIN)/DELTA_A);
          bk = (int)((rotated.b - B_MIN)/DELTA_B);
          for (lk = startL; lk <= endL; lk++)
          {
            if (interval_l[lk] + 0.5*(interval_l[lk + 1] - interval_l[lk]) > l
              break;
          }
          fit(l1, place.a, place.b, &ck, &mk, &yk);
          if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
              && yk >= 0 && yk <= 64
              && (inside_gamut[lk][ak][bk] == 1
                || (ak + 1 < A_STEPS
                  && inside_gamut[lk][ak + 1][bk] == 1)
                || (ak + 1 < A_STEPS && bk + 1 < B_STEPS
                  && inside_gamut[lk][ak + 1][bk + 1] == 1)
                || (bk + 1 < B_STEPS
                  && inside_gamut[lk][ak][bk + 1] == 1)
                || (ak - 1 >= 0 && bk + 1 < B_STEPS
                  && inside_gamut[lk][ak - 1][bk + 1] == 1)
                || (ak - 1 >= 0
                  && inside_gamut[lk][ak - 1][bk] == 1)
                || (ak - 1 >= 0 && bk - 1 >= 0
                  && inside_gamut[lk][ak - 1] [bk - 1] == 1)
                || (bk - 1 >= 0
                  && inside_gamut[lk][ak][bk - 1] == 1)
                || (ak + 1 < A_STEPS && bk - 1 >= 0
                  && inside_gamut[lk][ak + 1][bk - 1] == 1)))
            || inside_gamut[lk][ak][bk] == 1)
          {
            if (ck < 0 || ck > 64 || mk < 0 || mk > 64
              || yk < 0 || yk > 64)
              final_outside = 1;
            else
              final_outside = 0;
            final_l = l1;
            final_a = place.a;
            final_b = place.b;
            final_ah = ah;
            break;
          }
          else if (updown*(ls - l1) > 0)
          {
            final_outside = 0;
            final_l = l1;
            final_a = place.a;
```

```
            final_b = place.b;
            final_ah = ah;
            break;
          }
        }
      }
      if (final_outside == 1)
      {
        reached = 0;
        if (warped_ang0 < ANGLE_STEPS/4 || warped_ang0 >= (3*ANGLE_STEPS)/4)
        {
          for (ah = final_ah; ah >= CENTER_A; ah--)
          {
            place1.a = interval_a[ah];
            place1.b = plane_ratio*place1.a;
            rad1 = radius(place1);
            l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
            fit(l1, place1.a, place1.b, &ck, &mk, &yk);
            if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
              && yk >= 0 && yk <= 64) || (updown*(ls - l1) > 0))
            {
              reached = 1;
              a_width = 0.5*(final_a - place1.a);
              place.a = place1.a + a_width;
              break;
            }
          }
        }
        else
        {
          for (ah = final_ah; ah <= CENTER_A; ah++)
          {
            place1.a = interval_a[ah];
            place1.b = plane_ratio*place1.a;
            rad1 = radius(place1);
            l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
            fit(l1, place1.a, place1.b, &ck, &mk, &yk);
            if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
            && yk >= 0 && yk <= 64) || (updown*(ls - l1) > 0))
            {
              reached = 1;
              a_width = 0.5*(final_a - place1.a);
              place.a = place1.a + a_width;
              break;
            }
          }
        }
        if (reached == 0)
        {
          place.a = 0;
          place.b = 0;
          if (warped_ang0 < ANGLE_STEPS/4 || warped_ang0 >= (3*ANGLE_STEPS)/4
            a_width = max2(final_a, interval_a[CENTER_A + 1]);
          else
            a_width = min2(final_a, interval_a[CENTER_A - 1]);
        }
        count = 0;
        while (count < 11)
        {
          a_width = 0.5*a_width;
```

```
        place.b = plane_ratio*place.a;
        rad1 = radius(place);
        l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
        if (place.a == 0 && place.b == 0)
        {
          if (l0 < 30)
          {
            final_l = l1;
            final_a = 0;
            final_b = 0;
          }
          place.a = place.a + a_width;
        }
        else
        {
          fit(l1, place.a, place.b, &ck, &mk, &yk);
          if (ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
            && yk >= 0 && yk <= 64)
          {
            final_l = l1;
            final_a = place.a;
            final_b = place.b;
            place.a = place.a + a_width;
          }
          else if (updown*(ls - l1) > 0)
            place.a = place.a + a_width;
          else
            place.a = place.a - a_width;
        }
        count++;
      }
    }
    else
    {
      if (warped_ang0 < ANGLE_STEPS/4 || warped_ang0 >= (3*ANGLE_STEPS)/4)
      {
        for (ah = final_ah; ah < A_STEPS; ah++)
        {
          place1.a = interval_a[ah];
          place1.b = plane_ratio*place1.a;
          rad1 = radius(place1);
          if (rad1 > rad0)
          {
            a_width = 0.5*(place1.a - final_a);
            place.a = place1.a - a_width;
            break;
          }
          l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
          fit(l1, place1.a, place1.b, &ck, &mk, &yk);
          if ((ck < 0 || ck > 64 || mk < 0 || mk > 64
            || yk < 0 || yk > 64) || (updown*(ls - l1) > 0))
          {
            a_width = 0.5*(place1.a - final_a);
            place.a = place1.a - a_width;
            break;
          }
        }
      }
      else
      {
```

```
        for (ah = final_ah; ah >= 0; ah--)
        {
          place1.a = interval_a[ah];
          place1.b = plane_ratio*place1.a;
          rad1 = radius(place1);
          if (rad1 > rad0)
          {
            a_width = 0.5*(place1.a - final_a);
            place.a = place1.a - a_width;
            break;
          }
          l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
          fit(l1, place1.a, place1.b, &ck, &mk, &yk);
          if ((ck < 0 || ck > 64 || mk < 0 || mk > 64
            || yk < 0 || yk > 64) || (updown*(ls - l1) > 0))
          {
            a_width = 0.5*(place1.a - final_a);
            place.a = place1.a - a_width;
            break;
          }
        }
      }
      count = 0;
      while (count < 11)
      {
        a_width = 0.5*a_width;
        place.b = plane_ratio*place.a;
        rad1 = radius(place);
        l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
        fit(l1, place.a, place.b, &ck, &mk, &yk);
        if (ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
          && yk >= 0 && yk <= 64)
        {
          final_l = l1;
          final_a = place.a;
          final_b = place.b;
          place.a = place.a + a_width;
        }
        else if (updown*(ls - l1) > 0)
          place.a = place.a + a_width;
        else
          place.a = place.a - a_width;
        count++;
      }
    }
  }
  else
  {
    if (warped_ang0 == ANGLE_STEPS/4 || warped_ang0 == (3*ANGLE_STEPS)/4)
      plane_ratio = 0;
    else
    {
      plane_ratio = 1.0/tanpi((2.0*warped_ang0)/ANGLE_STEPS);
    }
    if (b1 > CENTER_B)
    {
      for (bh = b1; bh >= CENTER_B; bh--)
      {
        place.b = interval_b[bh];
        place.a = plane_ratio*place.b;
```

```
        rad1 = radius(place);
        if (rad1 > rad0)
          continue;
        l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
        rotate(ang0 - warped_ang0, place);
        ak = (int)((rotated.a - A_MIN)/DELTA_A);
        bk = (int)((rotated.b - B_MIN)/DELTA_B);
        for (lk = startL; lk <= endL; lk++)
        {
          if (interval_l[lk] + 0.5*(interval_l[lk + 1] - interval_l[lk]) > l
            break;
        }
        fit(l1, place.a, place.b, &ck, &mk, &yk);
        if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
            && yk >= 0 && yk <= 64
            && (inside_gamut[lk][ak][bk] == 1
              || (ak + 1 < A_STEPS
                && inside_gamut[lk][ak + 1][bk] == 1)
              || (ak + 1 < A_STEPS && bk + 1 < B_STEPS
                && inside_gamut[lk][ak + 1][bk + 1] == 1)
              || (bk + 1 < B_STEPS
                && inside_gamut[lk][ak][bk + 1] == 1)
              || (ak - 1 >= 0 && bk + 1 < B_STEPS
                && inside_gamut[lk][ak - 1][bk + 1] == 1)
              || (ak - 1 >= 0
                && inside_gamut[lk][ak - 1][bk] == 1)
              || (ak - 1 >= 0 && bk - 1 >= 0
                && inside_gamut[lk][ak - 1] [bk - 1] == 1)
              || (bk - 1 >= 0
                && inside_gamut[lk][ak][bk - 1] == 1)
              || (ak + 1 < A_STEPS && bk - 1 >= 0
                && inside_gamut[lk][ak + 1][bk - 1] == 1)))
          || inside_gamut[lk][ak][bk] == 1)
        {
          if (ck < 0 || ck > 64 || mk < 0 || mk > 64
            || yk < 0 || yk > 64)
            final_outside = 1;
          else
            final_outside = 0;
          final_l = l1;
          final_a = place.a;
          final_b = place.b;
          final_bh = bh;
          break;
        }
        else if (updown*(ls - l1) > 0)
        {
          final_outside = 0;
          final_l = l1;
          final_a = place.a;
          final_b = place.b;
          final_bh = bh;
          break;
        }
      }
    }
    else
    {
      for (bh = bl; bh <= CENTER_B; bh++)
      {
```

```
        place.b = interval_b[bh];
        place.a = plane_ratio*place.b;
        rad1 = radius(place);
        if (rad1 > rad0)
          continue;
        l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
        rotate(ang0 - warped_ang0, place);
        ak = (int)((rotated.a - A_MIN)/DELTA_A);
        bk = (int)((rotated.b - B_MIN)/DELTA_B);
        for (lk = startL; lk <= endL; lk++)
        {
          if (interval_l[lk] + 0.5*(interval_l[lk + 1] - interval_l[lk]) > -
            break;
        }
        fit(l1, place.a, place.b, &ck, &mk, &yk);
        if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
            && yk >= 0 && yk <= 64
            && (inside_gamut[lk][ak][bk] == 1
              || (ak + 1 < A_STEPS
                && inside_gamut[lk][ak + 1][bk] == 1)
              || (ak + 1 < A_STEPS && bk + 1 < B_STEPS
                && inside_gamut[lk][ak + 1][bk + 1] == 1)
              || (bk + 1 < B_STEPS
                && inside_gamut[lk][ak][bk + 1] == 1)
              || (ak - 1 >= 0 && bk + 1 < B_STEPS
                && inside_gamut[lk][ak - 1][bk + 1] == 1)
              || (ak - 1 >= 0
                && inside_gamut[lk][ak - 1][bk] == 1)
              || (ak - 1 >= 0 && bk - 1 >= 0
                && inside_gamut[lk][ak - 1] [bk - 1] == 1)
              || (bk - 1 >= 0
                && inside_gamut[lk][ak][bk - 1] == 1)
              || (ak + 1 < A_STEPS && bk - 1 >= 0
                && inside_gamut[lk][ak + 1][bk - 1] == 1)))
          || inside_gamut[lk][ak][bk] == 1)
        {
          if (ck < 0 || ck > 64 || mk < 0 || mk > 64
            || yk < 0 || yk > 64)
            final_outside = 1;
          else
            final_outside = 0;
          final_l = l1;
          final_a = place.a;
          final_b = place.b;
          final_bh = bh;
          break;
        }
        else if (updown*(ls - l1) > 0)
        {
          final_outside = 0;
          final_l = l1;
          final_a = place.a;
          final_b = place.b;
          final_bh = bh;
          break;
        }
      }
    }
    if (final_outside == 1)
    {
```

```
reached = 0;
if (warped_ang0 < ANGLE_STEPS/2)
{
  for (bh = final_bh; bh >= CENTER_B; bh--)
  {
    place1.b = interval_b[bh];
    place1.a = plane_ratio*place1.b;
    rad1 = radius(place1);
    l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
    fit(l1, place1.a, place1.b, &ck, &mk, &yk);
    if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
      && yk >= 0 && yk <= 64) || (updown*(ls - l1) > 0))
    {
      reached = 1;
      b_width = 0.5*(final_b - place1.b);
      place.b = place1.b + b_width;
      break;
    }
  }
}
else
{
  for (bh = final_bh; bh <= CENTER_B; bh++)
  {
    place1.b = interval_b[bh];
    place1.a = plane_ratio*place1.b;
    rad1 = radius(place1);
    l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
    fit(l1, place1.a, place1.b, &ck, &mk, &yk);
    if ((ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
    && yk >= 0 && yk <= 64) || (updown*(ls - l1) > 0))
    {
      reached = 1;
      b_width = 0.5*(final_b - place1.b);
      place.b = place1.b + b_width;
      break;
    }
  }
}
if (reached == 0)
{
  place.a = 0;
  place.b = 0;
  if (warped_ang0 < ANGLE_STEPS/2)
    b_width = max2(final_b, interval_b[CENTER_B + 1]);
  else
    b_width = min2(final_b, interval_b[CENTER_B - 1]);
}
count = 0;
while (count < 11)
{
  b_width = 0.5*b_width;
  place.a = plane_ratio*place.b;
  rad1 = radius(place);
  l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
  if (place.a == 0 && place.b == 0)
  {
    if (l0 < 30)
    {
      final_l = l1;
```

```
            final_a = 0;
            final_b = 0;
          }
          place.b = place.b + b_width;
        }
        else
        {
          fit(l1, place.a, place.b, &ck, &mk, &yk);
          if (ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
            && yk >= 0 && yk <= 64)
          {
            final_l = l1;
            final_a = place.a;
            final_b = place.b;
            place.b = place.b + b_width;
          }
          else if (updown*(ls - l1) > 0)
            place.b = place.b + b_width;
          else
            place.b = place.b - b_width;
        }
        count++;
      }
    }
    else
    {
      if (warped_ang0 < ANGLE_STEPS/2)
      {
        for (bh = final_bh; bh < B_STEPS; bh++)
        {
          place1.b = interval_b[bh];
          place1.a = plane_ratio*place1.b;
          rad1 = radius(place1);
          if (rad1 > rad0)
          {
            b_width = 0.5*(place1.b - final_b);
            place.b = place1.b - b_width;
            break;
          }
          l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
          fit(l1, place1.a, place1.b, &ck, &mk, &yk);
          if ((ck < 0 || ck > 64 || mk < 0 || mk > 64
            || yk < 0 || yk > 64) || (updown*(ls - l1) > 0))
          {
            b_width = 0.5*(place1.b - final_b);
            place.b = place1.b - b_width;
            break;
          }
        }
      }
      else
      {
        for (bh = final_bh; bh >= 0; bh--)
        {
          place1.b = interval_b[bh];
          place1.a = plane_ratio*place1.b;
          rad1 = radius(place1);
          if (rad1 > rad0)
          {
            b_width = 0.5*(place1.b - final_b);
```

```
                place.b = place1.b - b_width;
                break;
              }
              l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
              fit(l1, place1.a, place1.b, &ck, &mk, &yk);
              if ((ck < 0 || ck > 64 || mk < 0 || mk > 64
                || yk < 0 || yk > 64) || (updown*(ls - l1) > 0))
              {
                b_width = 0.5*(place1.b - final_b);
                place.b = place1.b - b_width;
                break;
              }
            }
          }
          count = 0;
          while (count < 11)
          {
            b_width = 0.5*b_width;
            place.a = plane_ratio*place.b;
            rad1 = radius(place);
            l1 = l0 + min2(l_step, -updown*SPACE_RATIO*(rad0 - rad1));
            fit(l1, place.a, place.b, &ck, &mk, &yk);
            if (ck >= 0 && ck <= 64 && mk >= 0 && mk <= 64
              && yk >= 0 && yk <= 64)
            {
              final_l = l1;
              final_a = place.a;
              final_b = place.b;
              place.b = place.b + b_width;
            }
            else if (updown*(ls - l1) > 0)
              place.b = place.b + b_width;
            else
              place.b = place.b - b_width;
            count++;
          }
        }
      }
      *l_ptr = final_l;
      *a_ptr = final_a;
      *b_ptr = final_b;
    }


    main()
    {

      FILE *lab_borders_ptr, *lut_ptr, *border_colors_ptr;
      FILE *m_minB_ptr, *m_maxB_ptr;
      int min, max, first, last, index, sign;
      int li, ai, bi, lj, aj, bj, ll, aa, bb;
      int ang, ang1, ang2, ang_diff;
      int ci, mi, yi, ki, cj, mj, kj, cc, mm, yy, kk;
      int h, hj, hh;
      int final_ci, final_mi, final_yi, final_ai, final_bi;
      int lowL, mediumL, highL;
      int cyan, magenta, yellow;
      int start_ai[ANGLE_STEPS], end_ai[ANGLE_STEPS];
      int start_bi[ANGLE_STEPS], end_bi[ANGLE_STEPS];
```

```
int close_a[8], close_b[8], far_a[16], far_b[16];
int border_a[24], border_b[24];
int startA[L_STEPS], endA[L_STEPS], startB[L_STEPS], endB[L_STEPS];
int minB[L_STEPS][A_STEPS], maxB[L_STEPS][A_STEPS];
int new_minB[L_STEPS][A_STEPS], new_maxB[L_STEPS][A_STEPS];
int m_minB[L_STEPS][A_STEPS], m_maxB[L_STEPS][A_STEPS];
double left, right, down, up;
double saturation, new_saturation, factor;
double l, a, b, lm, ln, an, bn, rad, radm, ratio, delta_l;
double warped_ang, double_ang;
cmy hue[16], saturation_border_color[ANGLE_STEPS];
a_b point, saturation_border[ANGLE_STEPS];
a_b outer_border[L_STEPS][ANGLE_STEPS];

if ((m_minB_ptr = fopen("m_minB", "r")) == NULL)        /* opens files */
{
  printf("cannot open m_minB\n");
  exit(1);
}
if ((m_maxB_ptr = fopen("m_maxB", "r")) == NULL)
{
  printf("cannot open m_maxB\n");
  exit(1);
}
if ((lab_borders_ptr = fopen(lab_borders, "wb")) == NULL)
{
  printf("cannot create lab_borders\n");
  exit(1);
}
if ((lut_ptr = fopen(lut, "wb")) == NULL)
{
  printf("cannot create lut\n");
  exit(1);
}
if ((border_colors_ptr = fopen(border_colors, "wb")) == NULL)
{
  printf("cannot create border_colors\n");
  exit(1);
}

for (li = 0; li < L_STEPS; li++)                        /* reads files */
  for (ai = 0; ai < A_STEPS; ai++)
{
  m_minB[li][ai] = B_STEPS;
  m_maxB[li][ai] = -1;
}
for (li = 0; li < L_STEPS; li++)
  for (ai = m_startA[li]; ai <= m_endA[li]; ai++)
{
  fread(&m_minB[li][ai], sizeof(int), 1, m_minB_ptr);
  fread(&m_maxB[li][ai], sizeof(int), 1, m_maxB_ptr);
}

min = (int)((dark - L_MIN)/DELTA_L);          /* computes the LABs of the
max = (int)((light - L_MIN)/DELTA_L);         /* interval centers */
lowL = min + 1 - extendL;
mediumL = min + 1 + extendL;
```

```
highL = max - 1 - (L_STEPS - max);
for (li = lowL; li < mediumL; li++)
  interval_l[li] = L_MIN + (min + 1 + (li - lowL)*0.5 + 0.25)*DELTA_L;
for (li = mediumL; li < highL; li++)
  interval_l[li] = L_MIN + (li + 0.5)*DELTA_L;
for (li = highL; li <= L_STEPS; li++)
  interval_l[li] = L_MIN + (highL + (li - highL)*0.5 + 0.25)*DELTA_L;
for (ai = 0; ai < A_STEPS; ai++)
  interval_a[ai] = A_MIN + (ai + 0.5)*DELTA_A;
for (bi = 0; bi < B_STEPS; bi++)
  interval_b[bi] = B_MIN + (bi + 0.5)*DELTA_B;


for (ang = 0; ang < ANGLE_STEPS; ang++)       /* computes warped angles */
  warped_angle[ang] = ang;
for (ang = 0; ang < 3; ang++)
  warped_angle[ANGLE_STEPS/4 - ang] = yellow_ang - 1.25*(ang + 1);
ang_diff = yellow_ang - ANGLE_STEPS/4;
for (ang = 1; ang <= ang_diff + yellow_shift; ang++)
  warped_angle[ANGLE_STEPS/4 + ang] = yellow_ang;
for (ang = 1; ang <= ANGLE_STEPS/24; ang++)
  warped_angle[ANGLE_STEPS/4 + ang_diff + yellow_shift + ang]
  = yellow_ang + 0.5*ang;
for (ang = 1; ang <= ANGLE_STEPS/30; ang++)
  warped_angle[ANGLE_STEPS/4 + ANGLE_STEPS/24
  + ang_diff + yellow_shift + ang]
  = yellow_ang + 0.5*ANGLE_STEPS/24 + 1.25*ang;
for (ang = 1; ang <= 2*(ang_diff + yellow_shift) + 5; ang++)
  warped_angle[ANGLE_STEPS/4 + ANGLE_STEPS/24 + ANGLE_STEPS/30
  + ang_diff + yellow_shift + ang]
  = yellow_ang + 0.5*ANGLE_STEPS/24 + ANGLE_STEPS/24
  + 1.5*ang;

ang1 = ANGLE_STEPS/36;
for (ang = 0; ang < ANGLE_STEPS/12; ang++)
  warped_angle[ang1 + ang] = ang1 + 0.5*ang;
ang1 = ANGLE_STEPS/24 + ANGLE_STEPS/36;
ang2 = ANGLE_STEPS/12 + ANGLE_STEPS/36;
for (ang = 0; ang < ANGLE_STEPS/18; ang++)
  warped_angle[ang2 + ang] = ang1 + 1.25*ang;
ang1 = ANGLE_STEPS/8 + ANGLE_STEPS/72;
ang2 = ANGLE_STEPS/6;
for (ang = 0; ang < ANGLE_STEPS/36; ang++)
  warped_angle[ang2 + ang] = ang1 + 1.5*ang;

ang1 = (5*ANGLE_STEPS)/8 + ANGLE_STEPS/18;
for (ang = 1; ang <= ANGLE_STEPS/12 + ANGLE_STEPS/18; ang++)
  warped_angle[ang1 - ang] = ang1 - 0.5*ang;
ang1 = ANGLE_STEPS/2 + ANGLE_STEPS/12 + ANGLE_STEPS/36;
ang2 = ANGLE_STEPS/2 + ANGLE_STEPS/24;
for (ang = 1; ang <= ANGLE_STEPS/18 + ANGLE_STEPS/72; ang++)
  warped_angle[ang2 - ang] = ang1 - 2.0*ang;

ang1 = (3*ANGLE_STEPS)/4 - ANGLE_STEPS/36;
for (ang = 0; ang < ANGLE_STEPS/12 + ANGLE_STEPS/36; ang++)
  warped_angle[ang1 + ang] = ang1 + 0.5*ang;
ang1 = (3*ANGLE_STEPS)/4 + ANGLE_STEPS/36;
ang2 = (3*ANGLE_STEPS)/4 + ANGLE_STEPS/12;
for (ang = 0; ang < ANGLE_STEPS/12 + ANGLE_STEPS/36; ang++)
  warped_angle[ang2 + ang] = ang1 + 1.25*ang;
```

```
ang1 = (3*ANGLE_STEPS)/4 + ANGLE_STEPS/6;
ang2 = (3*ANGLE_STEPS)/4 + ANGLE_STEPS/6 + ANGLE_STEPS/36;
for (ang = 0; ang < ANGLE_STEPS/18; ang++)
  warped_angle[ang2 + ang] = ang1 + 1.5*ang;

for (li = 0; li < L_STEPS; li++)                    /* initializes the LUT */
  for (ai = 0; ai < A_STEPS; ai++)
    for (bi = 0; bi < B_STEPS; bi++)
{
  LUT[li][ai][bi].c = (unsigned char)0;
  LUT[li][ai][bi].m = (unsigned char)0;
  LUT[li][ai][bi].y = (unsigned char)0;
  inside_gamut[li][ai][bi] = 0;
  concave[li][ai][bi] = 0;
}

for (li = lowL; li < L_STEPS; li++)                 /* sets the LUT CMYs */
  for (ai = 0; ai < A_STEPS; ai++)                  /* inside printer gamut */
    for (bi = 0; bi < B_STEPS; bi++)
{
  l = interval_l[li];
  a = interval_a[ai];
  b = interval_b[bi];
  if (ai != CENTER_A || bi != CENTER_B)
  {
    ang = angle(ai, bi);
    warped_ang = warped_angle[ang];
    point.a = a;
    point.b = b;
    rotate(warped_ang - ang, point);
    a = rotated.a;
    b = rotated.b;
  }
  fit(l, a, b, &ci, &mi, &yi);
  if (ci >= 0 && ci <= 64 && mi >= 0 && mi <= 64 && yi >= 0 && yi <= 64)
  {
    convert(ci, mi, yi);
    LUT[li][ai][bi].c = cmy_vector.c;
    LUT[li][ai][bi].m = cmy_vector.m;
    LUT[li][ai][bi].y = cmy_vector.y;
    inside_gamut[li][ai][bi] = 1;
  }
}

ai = CENTER_A;                                      /* resets the LUT CMYs */
bi = CENTER_B;                                      /* at the L-axis */
for (li = 0; li < L_STEPS; li++)
{
  ki = black[li];
  if (li >= lowL)
    inside_gamut[li][ai][bi] = 1;
  LUT[li][ai][bi].c = LUT[li][ai][bi].m = LUT[li][ai][bi].y
    = (unsigned char)ki;
}

startL = L_STEPS;                          /* finds the printer gamut borders
```

```
endL = -1;
for (li = 0; li < L_STEPS; li++)
{
  startA[li] = A_STEPS;
  endA[li] = -1;
}
for (li = 0; li < L_STEPS; li++)
{
  for (ai = 0; ai < A_STEPS; ai++)
  {
    minB[li][ai] = B_STEPS;
    maxB[li][ai] = -1;
  }
}
for (li = 0; li < L_STEPS; li++)
{
  for (ai = 0; ai < A_STEPS; ai++)
  {
    for (bi = 0; bi < B_STEPS; bi++)
    {
      if (inside_gamut[li][ai][bi] == 1)
      {
        startL = li;
        break;
      }
    }
    if (startL == li)
      break;
  }
  if (startL == li)
    break;
}
for (li = L_STEPS - 1; li >= 0; li--)
{
  for (ai = 0; ai < A_STEPS; ai++)
  {
    for (bi = 0; bi < B_STEPS; bi++)
    {
      if (inside_gamut[li][ai][bi] == 1)
      {
        endL = li;
        break;
      }
    }
    if (endL == li)
      break;
  }
  if (endL == li)
    break;
}


for (li = startL; li <= endL; li++)
{
  for (ai = 0; ai < A_STEPS; ai++)
  {
    for (bi = 0; bi < B_STEPS; bi++)
    {
      if (inside_gamut[li][ai][bi] == 1)
      {
```

```
            startA[li] = ai;
            break;
          }
        }
        if (startA[li] == ai)
          break;
      }
      for (ai = A_STEPS - 1; ai >= 0; ai--)
      {
        for (bi = 0; bi < B_STEPS; bi++)
        {
          if (inside_gamut[li][ai][bi] == 1)
          {
            endA[li] = ai;
            break;
          }
        }
        if (endA[li] == ai)
          break;
      }

      for (ai = startA[li]; ai <= endA[li]; ai++)
      {
        for (bi = 0; bi < B_STEPS; bi++)
        {
          if (inside_gamut[li][ai][bi] == 1)
          {
            minB[li][ai] = bi;
            break;
          }
        }
        for (bi = B_STEPS - 1; bi >= 0; bi--)
        {
          if (inside_gamut[li][ai][bi] == 1)
          {
            maxB[li][ai] = bi;
            break;
          }
        }
      }
    }

    for (li = startL; li <= endL; li++)          /* makes the printer gamut
    {                                            /* connected */
      for (ai = 0; ai < A_STEPS; ai++)
      {
        new_minB[li][ai] = minB[li][ai];
        new_maxB[li][ai] = maxB[li][ai];
      }

      for (bi = minB[li][CENTER_A] + 1; bi < CENTER_B; bi++)
      {
        if (inside_gamut[li][CENTER_A][bi] == 0)
          new_minB[li][CENTER_A] = bi + 1;
      }
      for (bi = maxB[li][CENTER_A] - 1; bi > CENTER_B; bi--)
      {
```

```
  if (inside_gamut[li][CENTER_A][bi] == 0)
    new_maxB[li][CENTER_A] = bi - 1;
}

for (ai = CENTER_A + 1; ai <= endA[li]; ai++)
{
  for (bi = minB[li][ai] + 1; bi < maxB[li][ai]; bi++)
    if (inside_gamut[li][ai][bi] == 0)
      new_minB[li][ai] = bi + 1;
  for (bi = maxB[li][ai] - 1; bi > minB[li][ai]; bi--)
    if (inside_gamut[li][ai][bi] == 0)
      new_maxB[li][ai] = bi - 1;
  if (new_minB[li][ai] > new_maxB[li][ai])
    if (new_minB[li][ai] > new_maxB[li][ai - 1]
      || maxB[li][ai] < new_minB[li][ai - 1])
      new_minB[li][ai] = minB[li][ai];
    else if (minB[li][ai] > new_maxB[li][ai - 1]
      || new_maxB[li][ai] < new_minB[li][ai - 1])
      new_maxB[li][ai] = maxB[li][ai];
    else if (new_maxB[li][ai] - minB[li][ai]
        > maxB[li][ai] - new_minB[li][ai])
      new_minB[li][ai] = minB[li][ai];
    else if (new_maxB[li][ai] - minB[li][ai]
        < maxB[li][ai] - new_minB[li][ai])
      new_maxB[li][ai] = maxB[li][ai];
    else if (minB[li][ai] >= CENTER_B)
      new_minB[li][ai] = minB[li][ai];
    else
      new_maxB[li][ai] = maxB[li][ai];
  if ((new_minB[li][ai] > new_maxB[li][ai - 1] + 1)
    || (new_maxB[li][ai] < new_minB[li][ai - 1] - 1))
  {
    for (aj = ai; aj <= endA[li]; aj++)
    {
      new_minB[li][aj] = B_STEPS;
      new_maxB[li][aj] = -1;
    }
    break;
  }
}

for (ai = CENTER_A - 1; ai >= startA[li]; ai--)
{
  for (bi = minB[li][ai] + 1; bi < maxB[li][ai]; bi++)
    if (inside_gamut[li][ai][bi] == 0)
      new_minB[li][ai] = bi + 1;
  for (bi = maxB[li][ai] - 1; bi > minB[li][ai]; bi--)
    if (inside_gamut[li][ai][bi] == 0)
      new_maxB[li][ai] = bi - 1;
  if (new_minB[li][ai] > new_maxB[li][ai])
    if (new_minB[li][ai] > new_maxB[li][ai + 1]
      || maxB[li][ai] < new_minB[li][ai + 1])
      new_minB[li][ai] = minB[li][ai];
    else if (minB[li][ai] > new_maxB[li][ai + 1]
      || new_maxB[li][ai] < new_minB[li][ai + 1])
      new_maxB[li][ai] = maxB[li][ai];
    else if (new_maxB[li][ai] - minB[li][ai]
        > maxB[li][ai] - new_minB[li][ai])
```

```
            new_minB[li][ai] = minB[li][ai];
          else if (new_maxB[li][ai] - minB[li][ai]
              < maxB[li][ai] - new_minB[li][ai])
            new_maxB[li][ai] = maxB[li][ai];
          else if (minB[li][ai] >= CENTER_B)
            new_minB[li][ai] = minB[li][ai];
          else
            new_maxB[li][ai] = maxB[li][ai];

        if ((new_minB[li][ai] > new_maxB[li][ai + 1] + 1)
          || (new_maxB[li][ai] < new_minB[li][ai + 1] - 1))
        {
          for (aj = startA[li]; aj <= ai; aj++)
          {
            new_minB[li][aj] = B_STEPS;
            new_maxB[li][aj] = -1;
          }
          break;
        }
      }


      for (ai = startA[li]; ai <= endA[li]; ai++)
      {
        for (bi = minB[li][ai]; bi < new_minB[li][ai]; bi++)
          inside_gamut[li][ai][bi] = 0;
        for (bi = new_maxB[li][ai] + 1; bi <= maxB[li][ai]; bi++)
          inside_gamut[li][ai][bi] = 0;
        minB[li][ai] = new_minB[li][ai];
        maxB[li][ai] = new_maxB[li][ai];
      }


      for (ai = CENTER_A + 1; ai <= endA[li]; ai++)
      {
        if (new_minB[li][ai] == B_STEPS)
        {
          for (aj = ai; aj <= endA[li]; aj++)
            for (bj = minB[li][aj]; bj <= maxB[li][aj]; bj++)
              inside_gamut[li][aj][bj] = 0;
          endA[li] = ai - 1;
          break;
        }
      }
      for (ai = CENTER_A - 1; ai >= startA[li]; ai--)
      {
        if (new_minB[li][ai] == B_STEPS)
        {
          for (aj = ai; aj >= startA[li]; aj--)
            for (bj = minB[li][aj]; bj <= maxB[li][aj]; bj++)
              inside_gamut[li][aj][bj] = 0;
          startA[li] = ai + 1;
          break;
        }
      }
    }


    for (li = startL; li <= endL; li++)          /* makes the printer gamut
    {                                            /* radially convex */
```

```
l = interval_l[li];
for (ang = 0; ang < ANGLE_STEPS; ang++)
{
  end_ai[ang] = -1;
  end_bi[ang] = -1;
  warped_ang = warped_angle[ang];
  if (warped_ang < (ANGLE_STEPS)/8 ||
    warped_ang >= (7*ANGLE_STEPS)/8)
  {
    ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
    for (ai = CENTER_A + 1; ai < A_STEPS; ai++)
    {
      a = interval_a[ai];
      b = ratio*a;
      bi = (int)((b - B_MIN)/DELTA_B);
      if (inside_gamut[li][ai][bi] == 0)
      {
        start_ai[ang] = ai;
        break;
      }
    }
    for (ai = start_ai[ang] + 1; ai < A_STEPS; ai++)
    {
      a = interval_a[ai];
      b = ratio*a;
      bi = (int)((b - B_MIN)/DELTA_B);
      if (inside_gamut[li][ai][bi] == 1)
      {
        end_ai[ang] = ai - 1;
        break;
      }
    }
  }
  else if (warped_ang >= (3*ANGLE_STEPS)/8
    && warped_ang < (5*ANGLE_STEPS)/8)
  {
    ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
    for (ai = CENTER_A - 1; ai >= 0; ai--)
    {
      a = interval_a[ai];
      b = ratio*a;
      bi = (int)((b - B_MIN)/DELTA_B);
      if (inside_gamut[li][ai][bi] == 0)
      {
        start_ai[ang] = ai;
        break;
      }
    }
    for (ai = start_ai[ang] - 1; ai >= 0; ai--)
    {
      a = interval_a[ai];
      b = ratio*a;
      bi = (int)((b - B_MIN)/DELTA_B);
      if (inside_gamut[li][ai][bi] == 1)
      {
        end_ai[ang] = ai + 1;
        break;
      }
    }
  }
```

```
else if (warped_ang >= (ANGLE_STEPS)/8
  && warped_ang < (3*ANGLE_STEPS)/8)
{
  if (warped_ang == ANGLE_STEPS/4)
    ratio = 0;
  else
    ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
  for (bi = CENTER_B + 1; bi < B_STEPS; bi++)
  {
    b = interval_b[bi];
    a = ratio*b;
    ai = (int)((a - A_MIN)/DELTA_A);
    if (inside_gamut[li][ai][bi] == 0)
    {
      start_bi[ang] = bi;
      break;
    }
  }
  for (bi = start_bi[ang] + 1; bi < B_STEPS; bi++)
  {
    b = interval_b[bi];
    a = ratio*b;
    ai = (int)((a - A_MIN)/DELTA_A);
    if (inside_gamut[li][ai][bi] == 1)
    {
      end_bi[ang] = bi - 1;
      break;
    }
  }
}
else
{
  if (warped_ang == (3*ANGLE_STEPS)/4)
    ratio = 0;
  else
  {
    ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
  }
  for (bi = CENTER_B - 1; bi >= 0; bi--)
  {
    b = interval_b[bi];
    a = ratio*b;
    ai = (int)((a - A_MIN)/DELTA_A);
    if (inside_gamut[li][ai][bi] == 0)
    {
      start_bi[ang] = bi;
      break;
    }
  }
  for (bi = start_bi[ang] - 1; bi >= 0; bi--)
  {
    b = interval_b[bi];
    a = ratio*b;
    ai = (int)((a - A_MIN)/DELTA_A);
    if (inside_gamut[li][ai][bi] == 1)
    {
      end_bi[ang] = bi + 1;
      break;
    }
  }
```

```
    }
  }

  for (ang = 0; ang < ANGLE_STEPS; ang++)
  {
    if (end_ai[ang] != -1 || end_bi[ang] != -1)
    {
      warped_ang = warped_angle[ang];
      if (warped_ang < (ANGLE_STEPS)/8
        || warped_ang >= (7*ANGLE_STEPS)/8)
      {
        ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
        for (ai = start_ai[ang]; ai <= end_ai[ang]; ai++)
        {
          a = interval_a[ai];
          b = ratio*a;
          bi = (int)((b - B_MIN)/DELTA_B);
          down = interval_b[bi] - 0.5*DELTA_B;
          up = interval_b[bi] + 0.5*DELTA_B;
          left = ratio*(a - 0.5*DELTA_A);
          if (warped_ang < (ANGLE_STEPS)/8)
          {
            if (down - left > -0.001)
              first = bi - 1;
            else
              first = bi;
          }
          else
          {
            if (left - up > -0.001)
              last = bi + 1;
            else
              last = bi;
          }
          right = ratio*(a + 0.5*DELTA_A);
          if (warped_ang < (ANGLE_STEPS)/8)
          {
            if (right - up > -0.001)
              last = bi + 1;
            else
              last = bi;
          }
          else
          {
            if (down - right > -0.001)
              first = bi - 1;
            else
              first = bi;
          }
          for (bj = first; bj <= last; bj++)
          {
            b = interval_b[bj];
            if (inside_gamut[li][ai][bj] == 0)
            {
              fit(l, a, b, &ci, &mi, &yi);
              ci = suckin(ci);
              mi = suckin(mi);
              yi = suckin(yi);
              convert(ci, mi, yi);
```

```
                LUT[li][ai][bj].c = cmy_vector.c;
                LUT[li][ai][bj].m = cmy_vector.m;
                LUT[li][ai][bj].y = cmy_vector.y;
                inside_gamut[li][ai][bj] = 1;
                concave[li][ai][bj] = 1;
              }
            }
          }
        }
        else if (warped_ang >= (3*ANGLE_STEPS)/8
          && warped_ang < (5*ANGLE_STEPS)/8)
        {
          ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
          for (ai = start_ai[ang]; ai >= end_ai[ang]; ai--)
          {
            a = interval_a[ai];
            b = ratio*a;
            bi = (int)((b - B_MIN)/DELTA_B);
            down = interval_b[bi] - 0.5*DELTA_B;
            up = interval_b[bi] + 0.5*DELTA_B;
            left = ratio*(a - 0.5*DELTA_A);
            if (warped_ang >= (ANGLE_STEPS)/2)
            {
              if (down - left > -0.001)
                first = bi - 1;
              else
                first = bi;
            }
            else
            {
              if (left - up > -0.001)
                last = bi + 1;
              else
                last = bi;
            }
            right = ratio*(a + 0.5*DELTA_A);
            if (warped_ang >= (ANGLE_STEPS)/2)
            {
              if (right - up > -0.001)
                last = bi + 1;
              else
                last = bi;
            }
            else
            {
              if (down - right > -0.001)
                first = bi - 1;
              else
                first = bi;
            }
            for (bj = first; bj <= last; bj++)
            {
              b = interval_b[bj];
              if (inside_gamut[li][ai][bj] == 0)
              {
                fit(l, a, b, &ci, &mi, &yi);
                ci = suckin(ci);
                mi = suckin(mi);
                yi = suckin(yi);
                convert(ci, mi, yi);
```

```
                LUT[li][ai][bj].c = cmy_vector.c;
                LUT[li][ai][bj].m = cmy_vector.m;
                LUT[li][ai][bj].y = cmy_vector.y;
                inside_gamut[li][ai][bj] = 1;
                concave[li][ai][bj] = 1;
              }
            }
          }
        }
        else if (warped_ang >= (ANGLE_STEPS)/8
          && warped_ang < (3*ANGLE_STEPS)/8)
        {
          if (warped_ang == ANGLE_STEPS/4)
            ratio = 0;
          else
            ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
          for (bi = start_bi[ang]; bi <= end_bi[ang]; bi++)
          {
            b = interval_b[bi];
            a = ratio*b;
            ai = (int)((a - A_MIN)/DELTA_A);
            down = interval_a[ai] - 0.5*DELTA_A;
            up = interval_a[ai] + 0.5*DELTA_A;
            left = ratio*(b - 0.5*DELTA_B);
            if (warped_ang <= (ANGLE_STEPS)/4)
            {
              if (down - left > -0.001)
                first = ai - 1;
              else
                first = ai;
            }
            else
            {
              if (left - up > -0.001)
                last = ai + 1;
              else
                last = ai;
            }
            right = ratio*(b + 0.5*DELTA_B);
            if (warped_ang <= (ANGLE_STEPS)/4)
            {
              if (right - up > -0.001)
                last = ai + 1;
              else
                last = ai;
            }
            else
            {
              if (down - right > -0.001)
                first = ai - 1;
              else
                first = ai;
            }
            for (aj = first; aj <= last; aj++)
            {
              a = interval_a[aj];
              if (inside_gamut[li][aj][bi] == 0)
              {
                fit(l, a, b, &ci, &mi, &yi);
                ci = suckin(ci);
```

```
                mi = suckin(mi);
                yi = suckin(yi);
                convert(ci, mi, yi);
                LUT[li][aj][bi].c = cmy_vector.c;
                LUT[li][aj][bi].m = cmy_vector.m;
                LUT[li][aj][bi].y = cmy_vector.y;
                inside_gamut[li][aj][bi] = 1;
                concave[li][aj][bi] = 1;
              }
            }
          }
        }
        else
        {
          if (warped_ang == (3*ANGLE_STEPS)/4)
            ratio = 0;
          else
          {
            ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
          }
          for (bi = start_bi[ang]; bi >= end_bi[ang]; bi--)
          {
            b = interval_b[bi];
            a = ratio*b;
            ai = (int)((a - A_MIN)/DELTA_A);
            down = interval_a[ai] - 0.5*DELTA_A;
            up = interval_a[ai] + 0.5*DELTA_A;
            left = ratio*(b - 0.5*DELTA_B);
            if (warped_ang <= (3*ANGLE_STEPS)/4)
            {
              if (down - left > -0.001)
                first = ai - 1;
              else
                first = ai;
            }
            else
            {
              if (left - up > -0.001)
                last = ai + 1;
              else
                last = ai;
            }
            right = ratio*(b + 0.5*DELTA_B);
            if (warped_ang <= (3*ANGLE_STEPS)/4)
            {
              if (right - up > -0.001)
                last = ai + 1;
              else
                last = ai;
            }
            else
            {
              if (down - right > -0.001)
                first = ai - 1;
              else
                first = ai;
            }
            for (aj = first; aj <= last; aj++)
            {
              a = interval_a[aj];
```

```
                if (inside_gamut[li][aj][bi] == 0)
                {
                  fit(l, a, b, &ci, &mi, &yi);
                  ci = suckin(ci);
                  mi = suckin(mi);
                  yi = suckin(yi);
                  convert(ci, mi, yi);
                  LUT[li][aj][bi].c = cmy_vector.c;
                  LUT[li][aj][bi].m = cmy_vector.m;
                  LUT[li][aj][bi].y = cmy_vector.y;
                  inside_gamut[li][aj][bi] = 1;
                  concave[li][aj][bi] = 1;
                }
              }
            }
          }
        }
      }


      for (ai = startA[li]; ai <= endA[li]; ai++)
      {
        minB[li][ai] = B_STEPS;
        maxB[li][ai] = -1;
        for (bi = 0; bi < B_STEPS; bi++)
        {
          if (inside_gamut[li][ai][bi] == 1)
          {
            minB[li][ai] = bi;
            break;
          }
        }
        for (bi = B_STEPS - 1; bi >= 0; bi--)
        {
          if (inside_gamut[li][ai][bi] == 1)
          {
            maxB[li][ai] = bi;
            break;
          }
        }
      }
      for (ai = startA[li]; ai <= endA[li]; ai++)
        for (bi = minB[li][ai] + 1; bi <= maxB[li][ai] - 1; bi++)
      {
        if (inside_gamut[li][ai][bi] == 0)
        {
          a = interval_a[ai];
          b = interval_b[bi];
          fit(l, a, b, &ci, &mi, &yi);
          ci = suckin(ci);
          mi = suckin(mi);
          yi = suckin(yi);
          convert(ci, mi, yi);
          LUT[li][ai][bi].c = cmy_vector.c;
          LUT[li][ai][bi].m = cmy_vector.m;
          LUT[li][ai][bi].y = cmy_vector.y;
          inside_gamut[li][ai][bi] = 1;

        }
      }
```

```
}

for (li = startL; li <= endL; li++)          /* extends the printer border *
{
  for (ai = CENTER_A - 2; ai <= CENTER_A + 2; ai++)
  {
    if (ai < startA[li] || ai > endA[li])
    {
      minB[li][ai] = CENTER_B - 2;
      maxB[li][ai] = CENTER_B + 2;
    }
    else
    {
      if (CENTER_B - 2 < minB[li][ai])
        minB[li][ai] = CENTER_B - 2;
      if (CENTER_B + 2 > maxB[li][ai])
        maxB[li][ai] = CENTER_B + 2;
    }
  }
  if (CENTER_A - 2 < startA[li])
    startA[li] = CENTER_A - 2;
  if (CENTER_A + 2 > endA[li])
    endA[li] = CENTER_A + 2;
}
for (li = startL; li <= endL; li++)
{
  for (ai = 0; ai < startA[li]; ai++)
  {
    minB[li][ai] = B_STEPS;
    maxB[li][ai] = -1;
  }
  for (ai = endA[li] + 1; ai < A_STEPS; ai++)
  {
    minB[li][ai] = B_STEPS;
    maxB[li][ai] = -1;
  }
}
for (li = startL; li <= endL; li++)
{
  startA[li] = startA[li] - PRINTER_PLUS;
  endA[li] = endA[li] + PRINTER_PLUS;
}
for (li = startL; li <= endL; li++)
{
  for (ai = startA[li]; ai <= endA[li]; ai++)
  {
    new_minB[li][ai] = minB[li][ai] - PRINTER_PLUS;
    new_maxB[li][ai] = maxB[li][ai] + PRINTER_PLUS;
    for (aj = ai - PRINTER_PLUS; aj <= ai + PRINTER_PLUS; aj++)
    {
      if (aj >= 0 && aj < A_STEPS)
      {
        new_minB[li][ai]
          = min2(new_minB[li][ai], minB[li][aj] - PRINTER_PLUS);
        new_maxB[li][ai]
          = max2(new_maxB[li][ai], maxB[li][aj] + PRINTER_PLUS);
      }
    }
  }
```

```
        for (ai = startA[li]; ai <= endA[li]; ai++)
        {
          minB[li][ai] = new_minB[li][ai];
          maxB[li][ai] = new_maxB[li][ai];
        }
      }
      for (li = startL; li <= endL; li++)
      {
        m_startA[li] = m_startA[li] - MONITOR_PLUS;
        m_endA[li] = m_endA[li] + MONITOR_PLUS;
      }
      for (li = startL; li <= endL; li++)
      {
        for (ai = m_startA[li]; ai <= m_endA[li]; ai++)
        {
          new_minB[li][ai] = m_minB[li][ai] - MONITOR_PLUS;
          new_maxB[li][ai] = m_maxB[li][ai] + MONITOR_PLUS;
          for (aj = ai - MONITOR_PLUS; aj <= ai + MONITOR_PLUS; aj++)
          {
            if (aj >= 0 && aj < A_STEPS)
            {
              new_minB[li][ai]
                = min2(new_minB[li][ai], m_minB[li][aj] - MONITOR_PLUS);
              new_maxB[li][ai]
                = max2(new_maxB[li][ai], m_maxB[li][aj] + MONITOR_PLUS);
            }
          }
        }
        for (ai = m_startA[li]; ai <= m_endA[li]; ai++)
        {
          m_minB[li][ai] = new_minB[li][ai];
          m_maxB[li][ai] = new_maxB[li][ai];
        }
      }
      for (li = startL; li <= endL; li++)
      {
        for (ai = 0; ai < A_STEPS; ai++)
        {
          minB[li][ai] = min2(minB[li][ai], m_minB[li][ai]);
          maxB[li][ai] = max2(maxB[li][ai], m_maxB[li][ai]);
        }
        for (ai = 0; ai < A_STEPS; ai++)
        {
          if (minB[li][ai] < B_STEPS && maxB[li][ai] > -1)
          {
            startA[li] = ai;
            break;
          }
        }
        for (ai = A_STEPS - 1; ai >= 0; ai--)
        {
          if (minB[li][ai] < B_STEPS && maxB[li][ai] > -1)
          {
            endA[li] = ai;
            break;
          }
        }
      }
      for (li = startL; li <= endL; li++)
        for (ang = 0; ang < ANGLE_STEPS; ang++)
```

```
{
  if ((ang < ANGLE_STEPS/8 || ang >= (7*ANGLE_STEPS)/8) ||
    (ang >= (3*ANGLE_STEPS)/8 && ang < (5*ANGLE_STEPS)/8))
  {
    ratio = tanpi((2.0*ang)/ANGLE_STEPS);
    if (ang < ANGLE_STEPS/4 || ang >= (3*ANGLE_STEPS)/4)
    {
      for (ai = A_STEPS - 1; ai >= CENTER_A; ai--)
      {
        a = interval_a[ai];
        b = ratio*a;
        bi = (int)((b - B_MIN)/DELTA_B);
        if (bi >= minB[li][ai] && bi <= maxB[li][ai])
        {
          outer_border[li][ang].a = a;
          outer_border[li][ang].b = b;
          break;
        }
      }
    }
    else
    {
      for (ai = 0; ai <= CENTER_A; ai++)
      {
        a = interval_a[ai];
        b = ratio*a;
        bi = (int)((b - B_MIN)/DELTA_B);
        if (bi >= minB[li][ai] && bi <= maxB[li][ai])
        {
          outer_border[li][ang].a = a;
          outer_border[li][ang].b = b;
          break;
        }
      }
    }
  }
  else
  {
    if (ang == ANGLE_STEPS/4 || ang == (3*ANGLE_STEPS)/4)
      ratio = 0;
    else
      ratio = 1.0/tanpi((2.0*ang)/ANGLE_STEPS);
    if (ang < ANGLE_STEPS/2)
    {
      for (bi = B_STEPS - 1; bi >= CENTER_B; bi--)
      {
        b = interval_b[bi];
        a = ratio*b;
        ai = (int)((a - A_MIN)/DELTA_A);
        if (bi >= minB[li][ai] && bi <= maxB[li][ai])
        {
          outer_border[li][ang].a = a;
          outer_border[li][ang].b = b;
          break;
        }
      }
    }
    else
    {
      for (bi = 0; bi <= CENTER_B; bi++)
```

```
        {
          b = interval_b[bi];
          a = ratio*b;
          ai = (int)((a - A_MIN)/DELTA_A);
          if (bi >= minB[li][ai] && bi <= maxB[li][ai])
          {
            outer_border[li][ang].a = a;
            outer_border[li][ang].b = b;
            break;
          }
        }
      }
    }
  }

for (ang = 0; ang < ANGLE_STEPS; ang++)        /* makes vertical outer */
{                                              /* border profiles monotonic
  if (ang < ANGLE_STEPS/4 || ang > (3*ANGLE_STEPS)/4)
    aa = 1;
  if (ang > ANGLE_STEPS/4 && ang < (3*ANGLE_STEPS)/4)
    aa = -1;
  if (ang == ANGLE_STEPS/4 || ang == (3*ANGLE_STEPS)/4)
    aa = 0;
  if (ang  > 0 && ang < ANGLE_STEPS/2)
    bb = 1;
  if (ang > ANGLE_STEPS/2 && ang < ANGLE_STEPS)
    bb = -1;
  if (ang == 0 || ang == ANGLE_STEPS/2)
    bb = 0;
  ll = saturation_l[ang];
  a = outer_border[ll][ang].a;
  b = outer_border[ll][ang].b;
  for (li = ll + 1; li <= endL; li++)
  {
    if (aa*outer_border[ll][ang].a > aa*a
      || bb*outer_border[ll][ang].b > bb*b)
    {
      outer_border[ll][ang].a = a;
      outer_border[ll][ang].b = b;
    }
    a = outer_border[ll][ang].a;
    b = outer_border[ll][ang].b;
  }
  a = outer_border[ll][ang].a;
  b = outer_border[ll][ang].b;
  for (li = ll - 1; li >= startL; li--)
  {
    if (aa*outer_border[ll][ang].a > aa*a
      || bb*outer_border[ll][ang].b > bb*b)
    {
      outer_border[ll][ang].a = a;
      outer_border[ll][ang].b = b;
    }
    a = outer_border[ll][ang].a;
    b = outer_border[ll][ang].b;
  }
}
```

```
for (ang = 0; ang < ANGLE_STEPS; ang++)          /* sets the saturation */
{                                                /* border LABs and CMYs */
  saturation = -1;
  warped_ang = warped_angle[ang];
  if (warped_ang < ANGLE_STEPS/8 || warped_ang >= (7*ANGLE_STEPS)/8)
  {
    ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
    point.a = interval_a[A_STEPS - 1];
    point.b = ratio*point.a;
  }
  else if (warped_ang >= (3*ANGLE_STEPS)/8 && warped_ang < (5*ANGLE_STEPS)
  {
    ratio = tanpi((2.0*warped_ang)/ANGLE_STEPS);
    point.a = interval_a[0];
    point.b = ratio*point.a;
  }
  else if (warped_ang >= ANGLE_STEPS/8 && warped_ang < (3*ANGLE_STEPS)/8)
  {
    if (warped_ang == ANGLE_STEPS/4)
      ratio = 0;
    else
      ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
    point.b = interval_b[B_STEPS - 1];
    point.a = ratio*point.b;
  }
  else
  {
    if (warped_ang == (3*ANGLE_STEPS)/4)
      ratio = 0;
    else
      ratio = 1.0/tanpi((2.0*warped_ang)/ANGLE_STEPS);
    point.b = interval_b[0];
    point.a = ratio*point.b;
  }
  for (li = startL; li <= endL; li++)
  {
    lm = interval_l[li];
    intersection(lm, ang, point, 0, &l, &a, &b);
    new_saturation = a*a + b*b;
    if (new_saturation > saturation)
    {
      ll = li;
      saturation = new_saturation;
    }
  }
  saturation = -1;
  if (ll == startL)
    lm = interval_l[ll];
  else
    lm = interval_l[ll - 1];
  delta_l = DELTA_L/64;
  if (ll == startL || ll == endL)
    last = 64;
  else
    last = 128;
  for (index = 1; index < 128; index++)
  {
    intersection(lm + index*delta_l, ang, point, 0, &l, &a, &b);
    new_saturation = a*a + b*b;
    if (new_saturation > saturation)
```

```
        {
          ln = l;
          an = a;
          bn = b;
          saturation = new_saturation;
        }
      }
      saturation_l[ang] = ln;
      saturation_border[ang].a = an;
      saturation_border[ang].b = bn;
      fit(ln, an, bn, &ci, &mi, &yi);
      final_ci = suckin(ci);
      final_mi = suckin(mi);
      final_yi = suckin(yi);
      convert(final_ci, final_mi, final_yi);
      saturation_border_color[ang].c = cmy_vector.c;
      saturation_border_color[ang].m = cmy_vector.m;
      saturation_border_color[ang].y = cmy_vector.y;
    }

    for (li = startL; li <= endL; li++)          /* sets the border CMYs */
    {
      lm = interval_l[li];
      for (ang = 0; ang < ANGLE_STEPS; ang++)
      {
        point.a = outer_border[li][ang].a;
        point.b = outer_border[li][ang].b;
        warped_ang = warped_angle[ang];
        rotate(warped_ang - ang, point);
        point.a = rotated.a;
        point.b = rotated.b;
        radm = radius(point);
        rad = radius(saturation_border[ang]);
        ln = saturation_l[ang];
        if (ln < lm)
          sign = 1;
        else if (ln > lm)
          sign = -1;
        else
          sign = 0;
        if (radm > rad && (sign*(lm - ln))/(radm - rad) < SPACE_RATIO)
        {
          border_color[li][ang].c = saturation_border_color[ang].c;
          border_color[li][ang].m = saturation_border_color[ang].m;
          border_color[li][ang].y = saturation_border_color[ang].y;
        }
        else
        {
          intersection(lm, ang, point, sign, &l, &a, &b);
          if (a == 0 && b == 0)
          {
            for (lj = startL; lj <= endL; lj++)
            {
              if (interval_l[lj] + (interval_l[lj + 1] - interval_l[lj])/2.0
                break;
            }
            if (lj == endL + 1)
              kj = 0;
            else
```

```
            kj = black[lj];
            border_color[li][ang].c = (unsigned char)kj;
            border_color[li][ang].m = (unsigned char)kj;
            border_color[li][ang].y = (unsigned char)kj;
          }
          else
          {
            fit(l, a, b, &ci, &mi, &yi);
            final_ci = suckin(ci);
            final_mi = suckin(mi);
            final_yi = suckin(yi);
            convert(final_ci, final_mi, final_yi);
            border_color[li][ang].c = cmy_vector.c;
            border_color[li][ang].m = cmy_vector.m;
            border_color[li][ang].y = cmy_vector.y;
          }
        }
      }
    }

    for (li = startL; li <= endL; li++)              /* converges the LUT into */
    {                                                /* the border */
      lm = interval_l[li];
      for (ai = startA[li]; ai <= endA[li]; ai++)
      {
        for (bi = minB[li][ai]; bi <= maxB[li][ai]; bi++)
        {
          point.a = interval_a[ai];
          point.b = interval_b[bi];
          saturation = radius(point);
          if (ai != CENTER_A || bi != CENTER_B)
          {
            ang = angle(ai, bi);
            warped_ang = warped_angle[ang];
            rotate(warped_ang - ang, point);
            point.a = rotated.a;
            point.b = rotated.b;
            if (inside_gamut[li][ai][bi] == 0)
            {
              radm = radius(point);
              rad = radius(saturation_border[ang]);
              ln = saturation_l[ang];
              if (ln < lm)
                sign = 1;
              else if (ln > lm)
                sign = -1;
              else
                sign = 0;
              if (radm > rad && (sign*(lm - ln))/(radm - rad) < SPACE_RATIO)
              {
                LUT[li][ai][bi].c = saturation_border_color[ang].c;
                LUT[li][ai][bi].m = saturation_border_color[ang].m;
                LUT[li][ai][bi].y = saturation_border_color[ang].y;
                transition[li][ai][bi] = 1;
              }
              else
              {
                intersection(lm, ang, point, sign, &l, &a, &b);
                if (a == 0 && b == 0)
```

```
                {
                  for (lj = startL; lj <= endL; lj++)
                  {
                    if (interval_l[lj] + DELTA_L/2.0 > l)
                      break;
                  }
                  if (lj == endL + 1)
                    kj = 0;
                  else
                    kj = black[lj];
                  LUT[li][ai][bi].c = (unsigned char)kj;
                  LUT[li][ai][bi].m = (unsigned char)kj;
                  LUT[li][ai][bi].y = (unsigned char)kj;
                  transition[li][ai][bi] = 1;
                }
                else
                {
                  fit(l, a, b, &ci, &mi, &yi);
                  final_ci = suckin(ci);
                  final_mi = suckin(mi);
                  final_yi = suckin(yi);
                  convert(final_ci, final_mi, final_yi);
                  LUT[li][ai][bi].c = cmy_vector.c;
                  LUT[li][ai][bi].m = cmy_vector.m;
                  LUT[li][ai][bi].y = cmy_vector.y;
                  transition[li][ai][bi] = 1;
                }
              }
            }
          }
        }
      }

      for (index = 0; index < 2; index++)                /* makes the colors near */
      {                                                  /* the gray axis more gray *
        close_a[index] = CENTER_A + 1;
        close_b[index] = CENTER_B + index;
        close_a[2 + index] = CENTER_A - index;
        close_b[2 + index] = CENTER_B + 1;
        close_a[4 + index] = CENTER_A - 1;
        close_b[4 + index] = CENTER_B - index;
        close_a[6 + index] = CENTER_A + index;
        close_b[6 + index] = CENTER_B - 1;
      }
      for (index = 0; index < 4; index++)
      {
        far_a[index] = CENTER_A + 2;
        far_b[index] = CENTER_B - 1 + index;
        far_a[4 + index] = CENTER_A + 1 - index;
        far_b[4 + index] = CENTER_B + 2;
        far_a[8 + index] = CENTER_A - 2;
        far_b[8 + index] = CENTER_B + 1 - index;
        far_a[12 + index] = CENTER_A - 1 + index;
        far_b[12 + index] = CENTER_B - 2;
      }
      for (index = 0; index < 5; index++)
      {
        border_a[index] = CENTER_A + 3;
```

```
        border_b[index] = CENTER_B - 2 + index;
        border_a[6 + index] = CENTER_A + 2 - index;
        border_b[6 + index] = CENTER_B + 3;
        border_a[12 + index] = CENTER_A - 3;
        border_b[12 + index] = CENTER_B + 2 - index;
        border_a[18 + index] = CENTER_A - 2 + index;
        border_b[18 + index] = CENTER_B - 3;
      }
      border_a[5] = CENTER_A + 3;
      border_b[5] = CENTER_B + 3;
      border_a[11] = CENTER_A - 3;
      border_b[11] = CENTER_B + 3;
      border_a[17] = CENTER_A - 3;
      border_b[17] = CENTER_B - 3;
      border_a[23] = CENTER_A + 3;
      border_b[23] = CENTER_B - 3;


      for (li = startL; li <= endL; li++)
      {
        l = interval_l[li];
        for (index = 0; index < 8; index++)
        {
          ai = border_a[3*index + 2];
          bi = border_b[3*index + 2];
          aa = far_a[2*index + 1];
          bb = far_b[2*index + 1];
          ang = angle(aa, bb);
          if (inside_gamut[li][ai][bi] == 1 || transition[li][ai][bi] == 1)
          {
            hue[2*index + 1].c = LUT[li][ai][bi].c;
            hue[2*index + 1].m = LUT[li][ai][bi].m;
            hue[2*index + 1].y = LUT[li][ai][bi].y;
          }
          else
          {
            hue[2*index + 1].c = border_color[li][ang].c;
            hue[2*index + 1].m = border_color[li][ang].m;
            hue[2*index + 1].y = border_color[li][ang].y;
          }
          kk = min3((int)hue[2*index + 1].c, hue[2*index + 1].m,
            hue[2*index + 1].y);
          if (ang > magenta_ang || ang < yellow_ang)
            hue[2*index + 1].c = (unsigned char)kk;
          if (ang > yellow_ang && ang < cyan_ang)
            hue[2*index + 1].m = (unsigned char)kk;
          if (ang > cyan_ang && ang < magenta_ang)
            hue[2*index + 1].y = (unsigned char)kk;
        }
        for (index = 0; index < 8; index++)
        {
          ai = border_a[3*index];
          bi = border_b[3*index];
          aj = border_a[3*index + 1];
          bj = border_b[3*index + 1];
          aa = far_a[2*index];
          bb = far_b[2*index];
          ang = angle(aa, bb);
          average2(li, ai, bi, aj, bj, 0);
          hue[2*index].c = averaged.c;
```

```
    hue[2*index].m = averaged.m;
    hue[2*index].y = averaged.y;
    kk = min3((int)hue[2*index].c, hue[2*index].m, hue[2*index].y);
    if (ang > magenta_ang || ang < yellow_ang)
      hue[2*index].c = (unsigned char)kk;
    if (ang > yellow_ang && ang < cyan_ang)
      hue[2*index].m = (unsigned char)kk;
    if (ang > cyan_ang && ang < magenta_ang)
      hue[2*index].y = (unsigned char)kk;
  }


ki = min3((int)LUT[li][CENTER_A][CENTER_B].c,
  (int)LUT[li][CENTER_A][CENTER_B].m, (int)LUT[li][CENTER_A][CENTER_B].y
for (index = 0; index < 8; index++)
{
  cc = (int)hue[2*index + 1].c;
  mm = (int)hue[2*index + 1].m;
  yy = (int)hue[2*index + 1].y;
  kk = min3((int)hue[2*index + 1].c, (int)hue[2*index + 1].m,
    (int)hue[2*index + 1].y);
  ci = (int)(ki + (1.0/3.0)*((double)(cc - ki)));
  mi = (int)(ki + (1.0/3.0)*((double)(mm - ki)));
  cj = (int)(ki + (2.0/3.0)*((double)(cc - ki)));
  mj = (int)(ki + (2.0/3.0)*((double)(mm - ki)));
  if (ki <= 1 && (ci == cj || mi == mj) && (cc != 0 || mm != 0))
    yi = (int)(ki + (2.0/3.0)*((double)(yy - ki)) + 0.5);
  else
    yi = (int)(ki + (1.0/3.0)*((double)(yy - ki)) + 0.5);
  if (ci == mi && mi == yi && ci != 0)
  {
    if (cc == kk)
      ci--;
    if (mm == kk)
      mi--;
    if (yy == kk)
      yi--;
  }
  kj = min3(ci, mi, yi);
  if (ki <= 1 && kj > kk)
  {
    if (cc == kk)
      ci = kk;
    if (mm == kk)
      mi = kk;
    if (yy == kk)
      yi = kk;
  }
  if (ci == 0 && mi == 0 && (ki != 0 || index != 2))
  {
    if (cc > mm)
      ci++;
    else if (cc < mm)
      mi++;
    else
    {
      ci++;
      mi++;
    }
  }
```

```
            ai = close_a[index];
            bi = close_b[index];
            LUT[li][ai][bi].c = (unsigned char)ci;
            LUT[li][ai][bi].m = (unsigned char)mi;
            LUT[li][ai][bi].y = (unsigned char)yi;
            inside_gamut[li][ai][bi] = 1;
          }

          for (index = 0; index < 16; index++)
          {
            cc = (int)hue[index].c;
            mm = (int)hue[index].m;
            yy = (int)hue[index].y;
            kk = min3((int)hue[index].c, (int)hue[index].m, (int)hue[index].y);
            ci = (int)(ki + (2.0/3.0)*((double)(cc - ki)));
            mi = (int)(ki + (2.0/3.0)*((double)(mm - ki)));
            yi = (int)(ki + (2.0/3.0)*((double)(yy - ki)) + 0.5);
            if (ci == mi && mi == yi && ci != 0)
            {
              if (cc == kk)
                ci--;
              if (mm == kk)
                mi--;
              if (yy == kk)
                yi--;
            }
            kj = min3(ci, mi, yi);
            if (ki <= 2 && kj > kk)
            {
              if (cc == kk)
                ci = kk;
              if (mm == kk)
                mi = kk;
              if (yy == kk)
                yi = kk;
            }
            if (ci == 0 && mi == 0 && (ki != 0 || (index != 5 && index != 6)))
            {
              if (cc > mm)
                ci++;
              else if (cc < mm)
                mi++;
              else
              {
                ci++;
                mi++;
              }
            }
            ai = far_a[index];
            bi = far_b[index];
            LUT[li][ai][bi].c = (unsigned char)ci;
            LUT[li][ai][bi].m = (unsigned char)mi;
            LUT[li][ai][bi].y = (unsigned char)yi;
            inside_gamut[li][ai][bi] = 1;
          }
        }

        for (index = 0; index < 5; index++)          /* recomputes colors in */
```

```
{                                                          /* concave areas */
  for (li = startL; li <= endL; li++)
    for (ai = startA[li]; ai <= endA[li]; ai++)
      for (bi = minB[li][ai]; bi <= maxB[li][ai]; bi++)
  {
    if (concave[li][ai][bi] == 1)
    {
      average4(li, ai, bi, 1);
      LUT[li][ai][bi].c = averaged.c;
      LUT[li][ai][bi].m = averaged.m;
      LUT[li][ai][bi].y = averaged.y;
    }
  }
}


for (li = startL; li <= endL; li++)                        /* computes startB, endB */
{
  startB[li] = minB[li][CENTER_A];
  endB[li] = maxB[li][CENTER_A];
  for (ai = startA[li]; ai <= endA[li]; ai++)
  {
    if (minB[li][ai] < startB[li])
      startB[li] = minB[li][ai];
    if (maxB[li][ai] > endB[li])
      endB[li] = maxB[li][ai];
  }
}


for (li = startL; li <= endL; li++)                        /* fills complete LUT */
{                                                          /* rectangles */
  for (ai = startA[li]; ai <= endA[li]; ai++)
  {
    for (bi = startB[li]; bi < minB[li][ai]; bi++)
    {
      ang = angle(ai, bi);
      LUT[li][ai][bi].c = border_color[li][ang].c;
      LUT[li][ai][bi].m = border_color[li][ang].m;
      LUT[li][ai][bi].y = border_color[li][ang].y;
    }
    for (bi = maxB[li][ai] + 1; bi <= endB[li]; bi++)
    {
      ang = angle(ai, bi);
      LUT[li][ai][bi].c = border_color[li][ang].c;
      LUT[li][ai][bi].m = border_color[li][ang].m;
      LUT[li][ai][bi].y = border_color[li][ang].y;
    }
  }
}


for (li = 0; li < startL; li++)                /* sets startA, endA, startB, endB *
{                                              /* and border CMYs for low Ls */
  startA[li] = endA[li] = CENTER_A;
  startB[li] = endB[li] = CENTER_B;
}
for (li = 0; li < startL; li++)
  for (ang = 0; ang < ANGLE_STEPS; ang++)
{
```

```
        ki = min3((int)LUT[li][CENTER_A][CENTER_B].c,
          LUT[li][CENTER_A][CENTER_B].m,
          LUT[li][CENTER_A][CENTER_B].y);
        border_color[li][ang].c = border_color[li][ang].m = border_color[li][ang
          = (unsigned char)ki;
      }

      fwrite(&startL, sizeof(int), 1,                    /* writes the lab_borders */
        lab_borders_ptr);
      fwrite(&endL, sizeof(int), 1, lab_borders_ptr);
      for (li = 0; li < L_STEPS; li++)
      {
        fwrite(&startA[li], sizeof(int), 1, lab_borders_ptr);
        fwrite(&endA[li], sizeof(int), 1, lab_borders_ptr);
      }
      for (li = 0; li < L_STEPS; li++)
      {
        fwrite(&startB[li], sizeof(int), 1, lab_borders_ptr);
        fwrite(&endB[li], sizeof(int), 1, lab_borders_ptr);
      }

      for (li = 0; li < L_STEPS; li++)                   /* writes the LUT */
        for (ai = startA[li]; ai <= endA[li]; ai++)
          for (bi = startB[li]; bi <= endB[li]; bi++)
        fwrite(&LUT[li][ai][bi], sizeof(cmy), 1, lut_ptr);

      for (li = 0; li < L_STEPS; li++)                   /* writes the border_colors
        for (ang = 0; ang < ANGLE_STEPS; ang++)
        fwrite(&border_color[li][ang], sizeof(cmy), 1, border_colors_ptr);
    }
```

Claims

1. A color printer look-up table for providing color primary values corresponding to colors in device-independent color space, the color printer table having smoothly varying color values for colors outside the color printer gamut.

2. A color printer look-up table according to Claim 1, wherein colors outside the color printer gamut have monotonically increasing lightness from the darkest value in the printer table to the lightest value in the printer table.

3. A color printer look-up table according to Claim 1 or 2, wherein the color primary values include color primary values for colors inside the color printer gamut and for colors outside the color printer gamut.

4. A color printer look-up table according to Claim 3, wherein the color primary values include color primary values for colors within a color monitor gamut.

5. A color printer look-up table according to Claim 3 or 4, wherein for colors inside the color printer gamut, the color primary values are provided in accurate correspondence to the associated colors.

6. A color printer look-up table according to Claim 5, wherein the color primary values outside the color printer gamut are provided so as to preserve hue, increase saturation, and change lightness.

7. A color printer driver comprising:

   a printer table for providing color primary values corresponding to colors within the color printer gamut and for colors outside the color printer gamut, the color primary values varying smoothly for colors

outside the color printer gamut;

means for accepting a command for printing a designated color; and

control means for extracting color primary values corresponding to the designated color from the printer table.

8. A color printer look-up table according to Claim 7, wherein colors outside the color printer gamut have monotonically increasing lightness from the darkest value in the printer table to the lightest value in the printer table.

9. A color printer driver according to Claim 7 or 8, wherein the printer table includes color primary values for colors within a color monitor gamut.

10. A color printer driver according to Claim 7, 8, 9 or 10 wherein for colors within the color printer gamut, the color primary values in the printer table are provided in accurate correspondence to the associated colors.

11. A color printer driver according to any of Claims 7 to 10, wherein the color primary values for colors in the printer table outside the printer gamut are provided so as to preserve hue, increase saturation, and change lightness.

12. A color printer driver according to any of Claims 7 to 11, further comprising a border table for providing color primary values for colors outside the printer table, wherein said control means selects the printer table or the border table based on the designated color and extracts color primary values from the selected one of the printer table and the border table.

13. A color printer driver according to Claim 12, wherein the border table is arranged in a wheel-like arrangement of cells centered on the lightness axis.

14. A color printer driver according to Claim 12 or 13, wherein the color primary values for colors in the border table outside the printer gamut are provided so as to preserve hue, increase saturation, and change lightness.

15. A method for building a printer table comprising:

determining a color printer gamut edge;

mapping color primary values into the printer table for colors within the color printer gamut edge; and

mapping transition colors into the printer table between the printer gamut edge and the edge of the printer table by constant angle extension to the edge of the printer gamut edge so as to increase color saturation.

16. A method according to Claim 15, further comprising the steps of mapping all points in a wedge subtended by the constant angle from the maximum saturation point of the printer gamut edge onto the maximum saturation point.

17. A method according to Claim 15 or 16, wherein the constant angle is substantially about 15°.

18. A method according to Claim 15, 16, 17 or 18, wherein the color primary values for transition colors in the printer table are provided for colors within a color monitor gamut.

19. A method according to any of Claims 15 to 18, wherein for colors within the color printer gamut, the color primary values are provided in accurate correspondence to the associated colors.

20. A method according to Claim 19, wherein the color primary values outside the printer gamut are provided so as to preserve hue, increase saturation, and change lightness.

21. A method according to any of Claims 15 to 20, further comprising the step of building a border table.

22. A method according to Claim 21, wherein the color primary values for colors in the border table outside the printer gamut are provided so as to preserve hue, increase saturation, and change lightness.

23. A method of color image reproduction wherein not all desired colors are available via a desired output de-

vice, and wherein unavailable colors are mapped to respective availabl colors so as to preserve monotonic increases in lightness of the desir d color.

24. A method of color image reproduction wherein not all desired colors are available via a desired output device, and wherein unavailable colors are mapped to respectiv available colors so as to preserve an angle in color space between the desired color and a plane of constant luminance.

25. An image signal or recording generated using a method or apparatus according to any preceding claim.

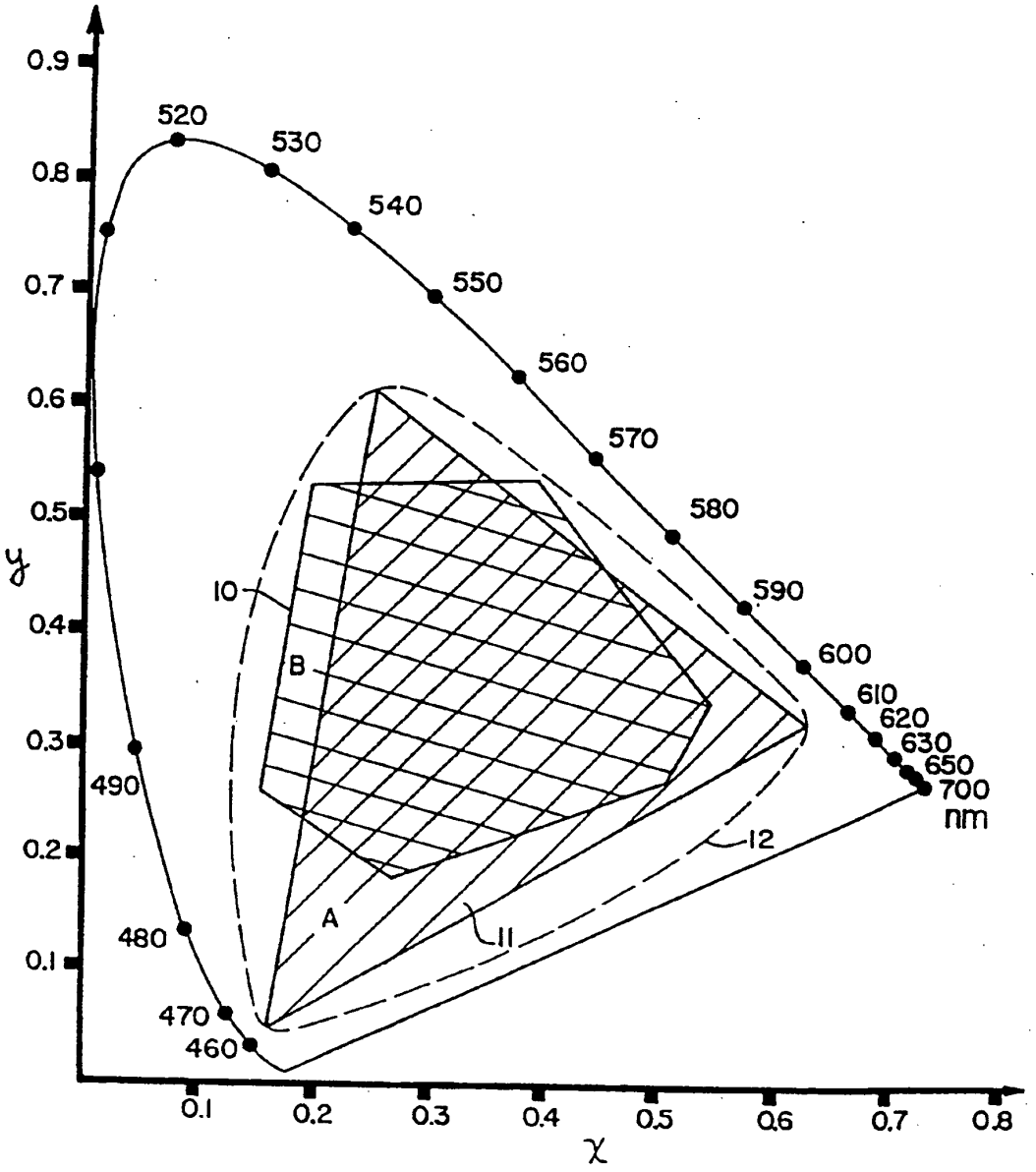
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
y
520
530
540
550
560
570
580
590
600
610
620
630
650
700
nm
490
480
470
460
10
B
A
11
12
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
χ

FIG 1

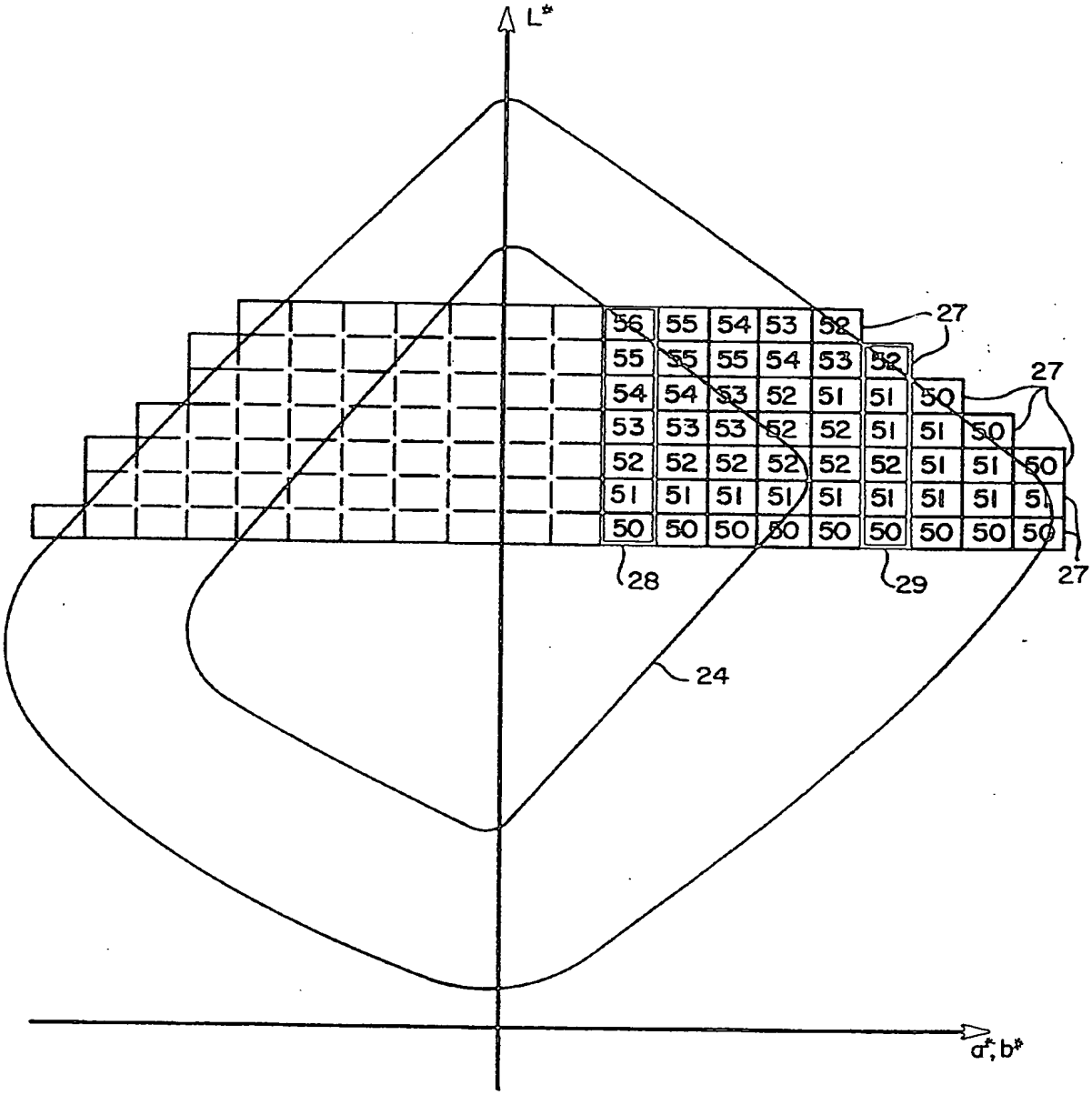
L*
56 55 54 53 52
55 55 55 54 53 52
54 54 53 52 51 51 50
53 53 53 52 52 51 51 50
52 52 52 52 52 52 51 51 50
51 51 51 51 51 51 51 51 51
50 50 50 50 50 50 50 50 50
27
27
27
28
29
24
a*, b*

FIG. 2

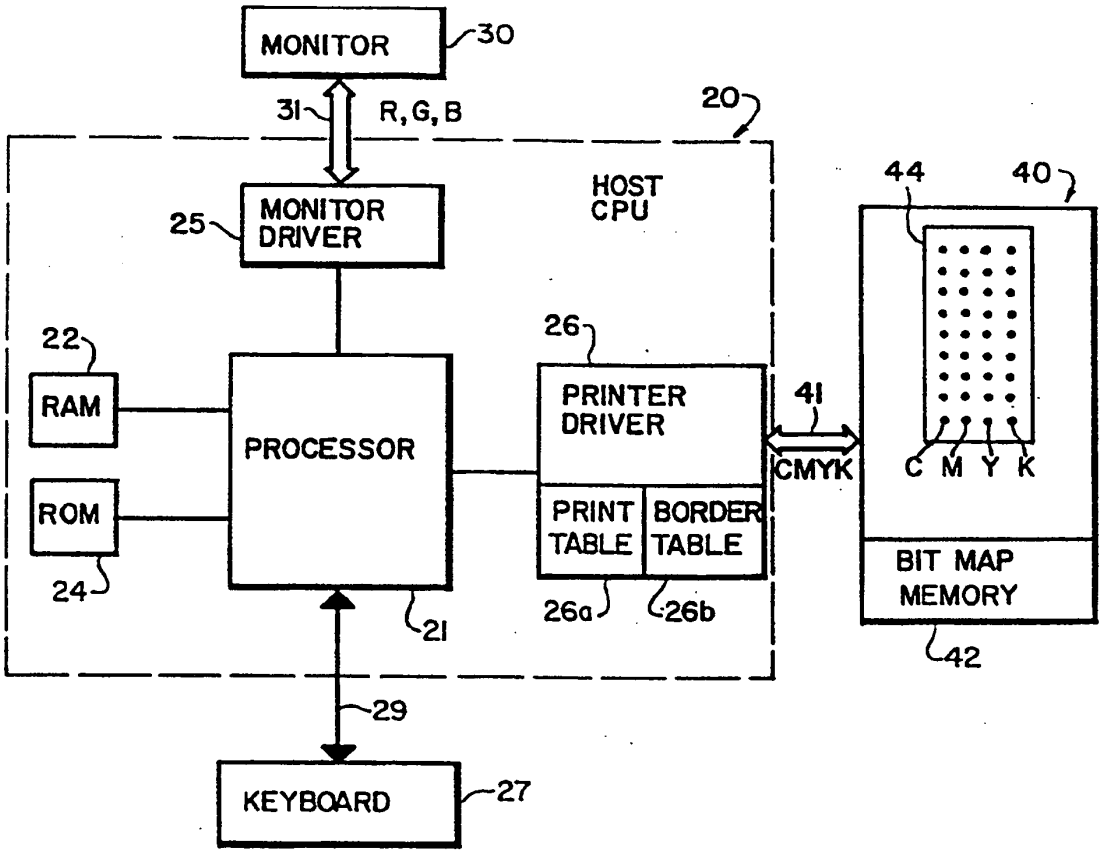
MONITOR
30
31
R, G, B
20
HOST CPU
MONITOR DRIVER
25
22
RAM
ROM
24
PROCESSOR
21
26
PRINTER DRIVER
PRINT TABLE
BORDER TABLE
26a
26b
41
CMYK
44
40
C M Y K
BIT MAP MEMORY
42
29
KEYBOARD
27

FIG. 3

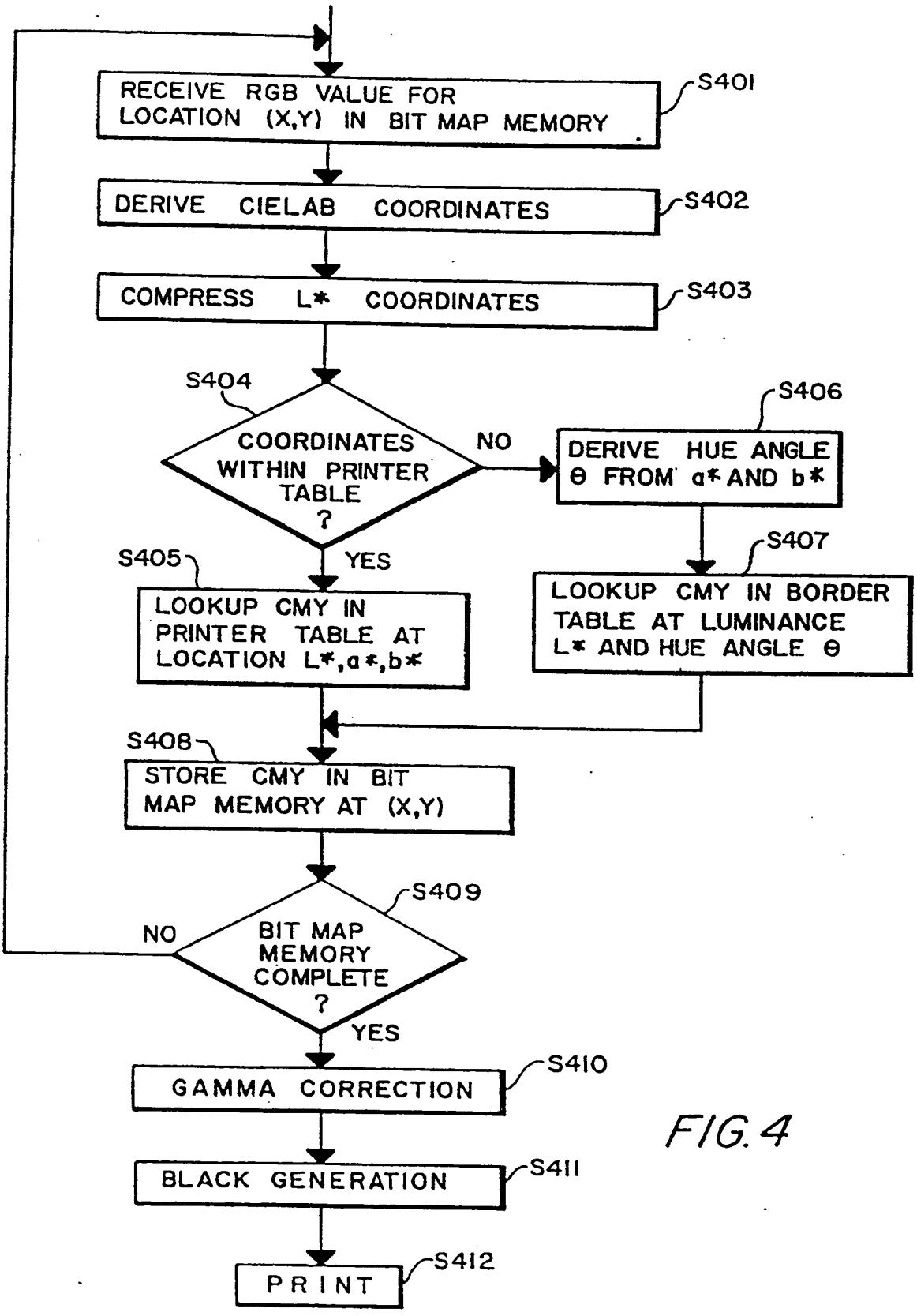
RECEIVE RGB VALUE FOR LOCATION (X,Y) IN BIT MAP MEMORY
S401
DERIVE CIELAB COORDINATES
S402
COMPRESS L* COORDINATES
S403
S404
COORDINATES WITHIN PRINTER TABLE ?
NO
YES
S406
DERIVE HUE ANGLE θ FROM a* AND b*
S407
S405
LOOKUP CMY IN PRINTER TABLE AT LOCATION L*,a*,b*
LOOKUP CMY IN BORDER TABLE AT LUMINANCE L* AND HUE ANGLE θ
S408
STORE CMY IN BIT MAP MEMORY AT (X,Y)
S409
NO
BIT MAP MEMORY COMPLETE ?
YES
S410
GAMMA CORRECTION
S411
BLACK GENERATION
S412
PRINT

FIG. 4

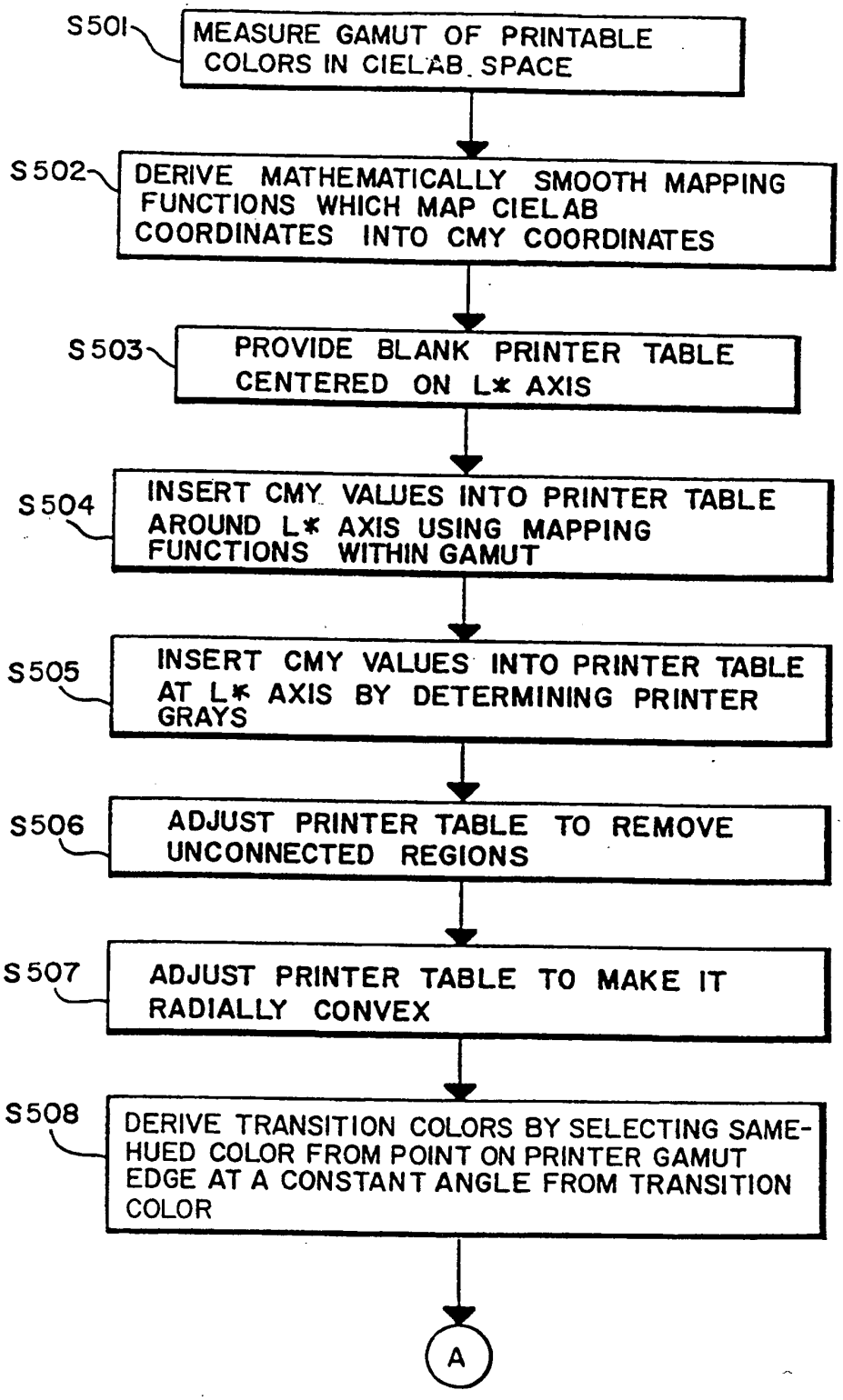
S501
MEASURE GAMUT OF PRINTABLE COLORS IN CIELAB SPACE
S502
DERIVE MATHEMATICALLY SMOOTH MAPPING FUNCTIONS WHICH MAP CIELAB COORDINATES INTO CMY COORDINATES
S503
PROVIDE BLANK PRINTER TABLE CENTERED ON L* AXIS
S504
INSERT CMY VALUES INTO PRINTER TABLE AROUND L* AXIS USING MAPPING FUNCTIONS WITHIN GAMUT
S505
INSERT CMY VALUES INTO PRINTER TABLE AT L* AXIS BY DETERMINING PRINTER GRAYS
S506
ADJUST PRINTER TABLE TO REMOVE UNCONNECTED REGIONS
S507
ADJUST PRINTER TABLE TO MAKE IT RADIALLY CONVEX
S508
DERIVE TRANSITION COLORS BY SELECTING SAME-HUED COLOR FROM POINT ON PRINTER GAMUT EDGE AT A CONSTANT ANGLE FROM TRANSITION COLOR
A

FIG.5(a)

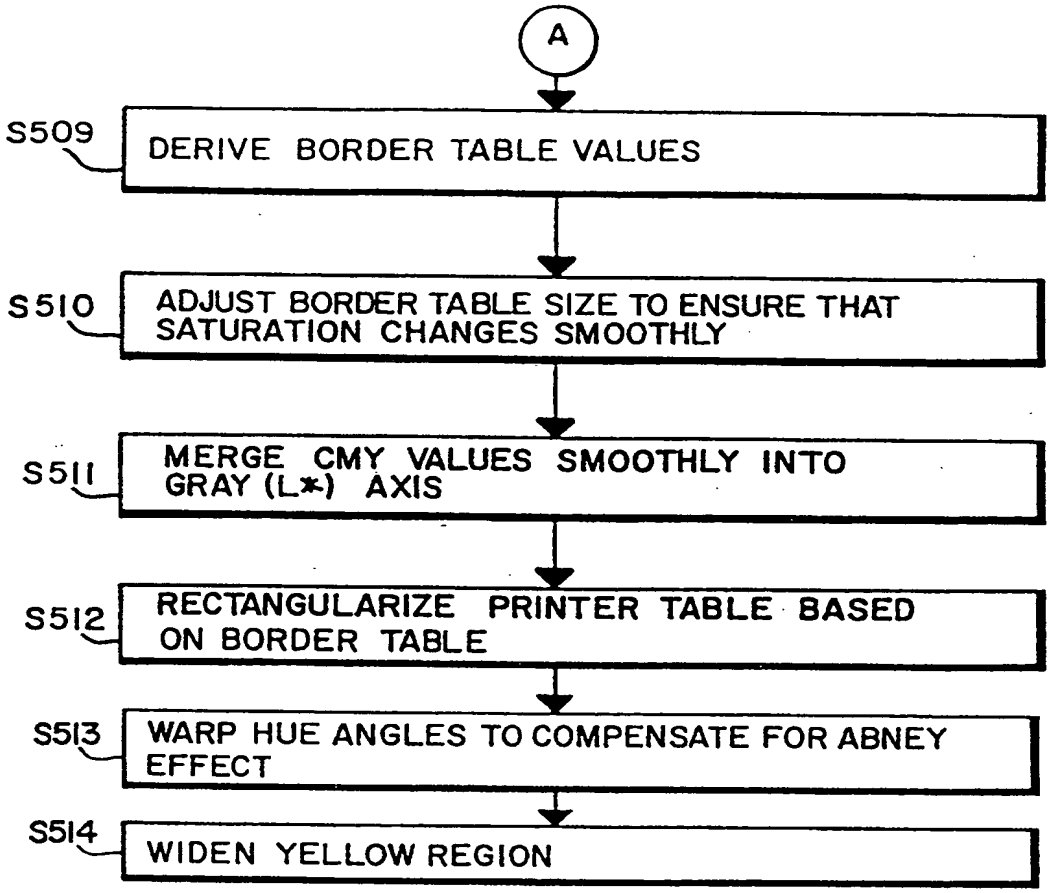

FIG. 5(b)

FIG. 5(a)
FIG. 5(b)

FIG. 5

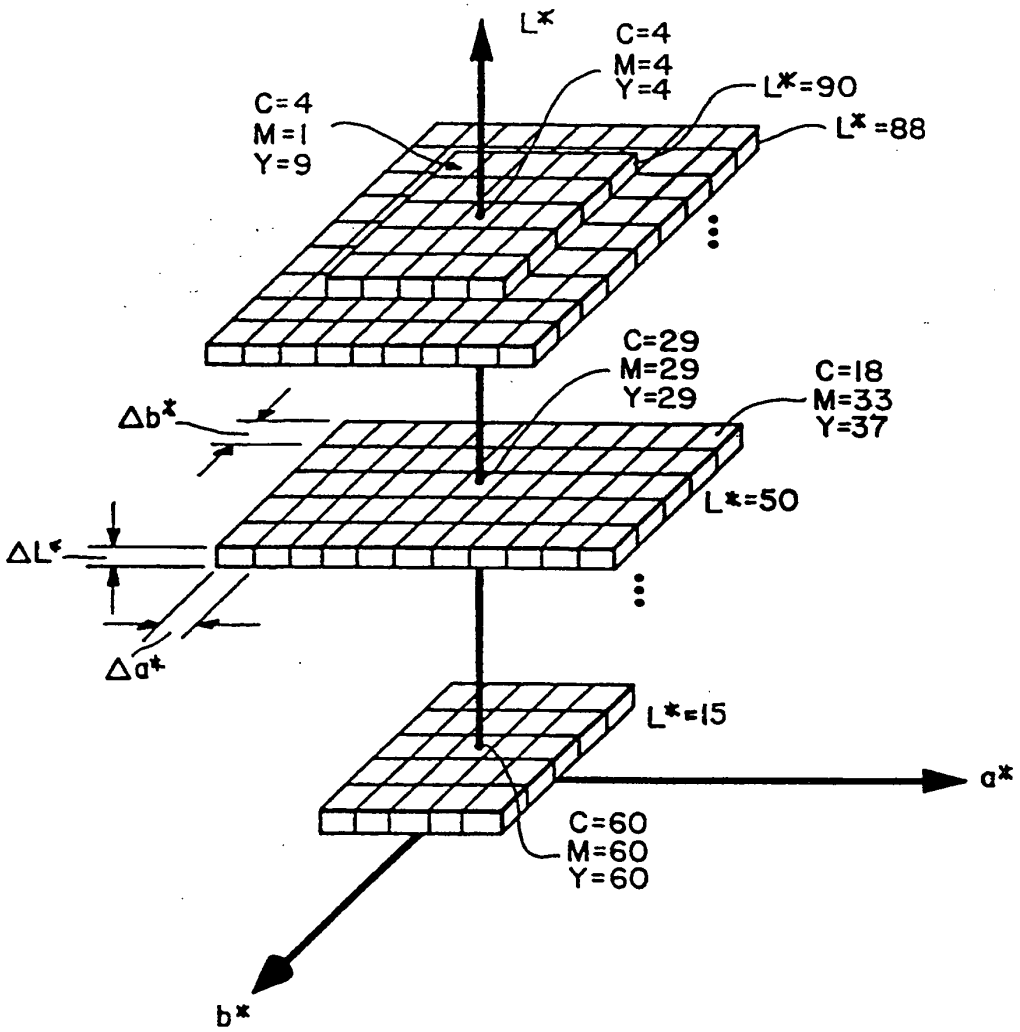
L*
C=4
M=4
Y=4
L*=90
L*=88
C=4
M=1
Y=9
C=29
M=29
Y=29
C=18
M=33
Y=37
Δb*
L*=50
ΔL*
Δa*
L*=15
a*
C=60
M=60
Y=60
b*

FIG. 6

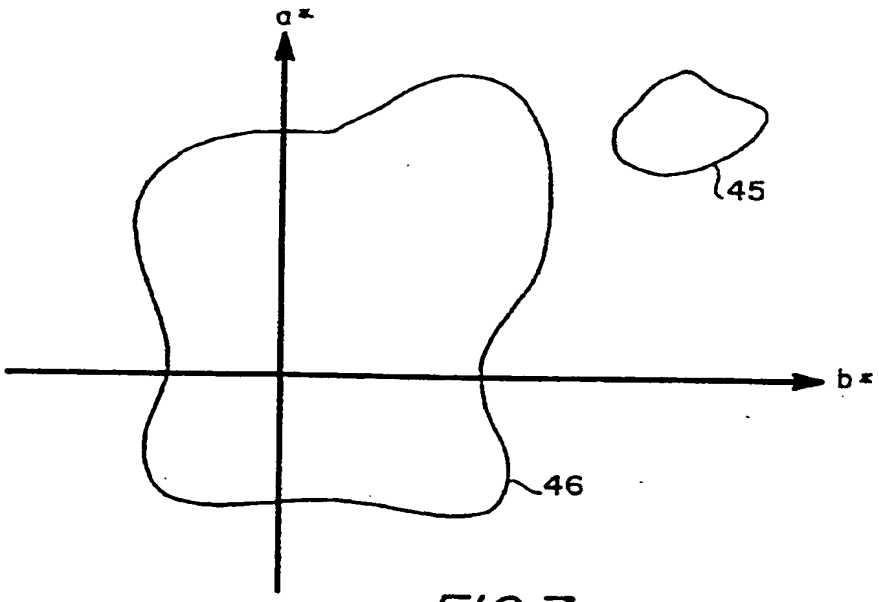

FIG. 7

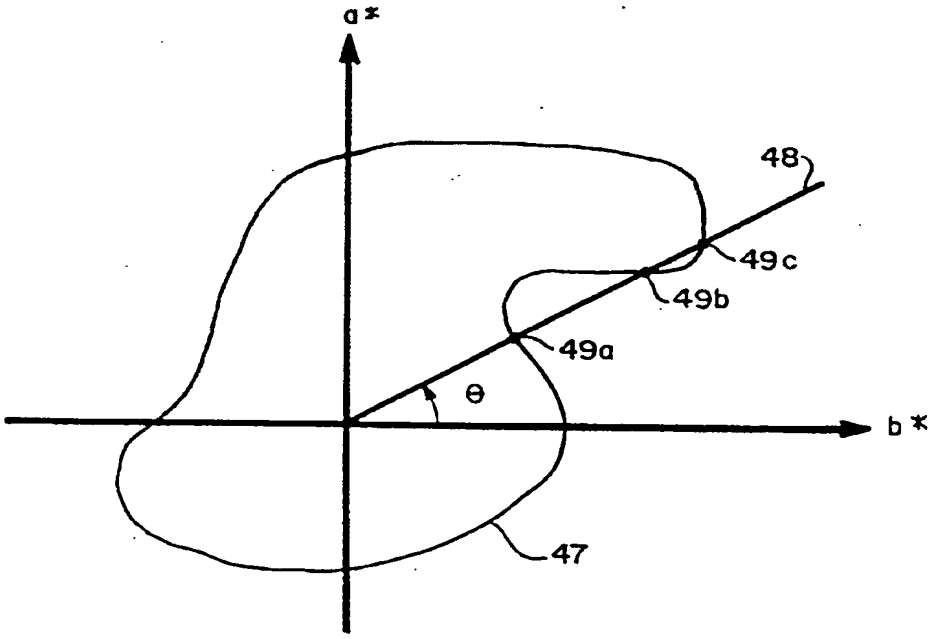

FIG. 8

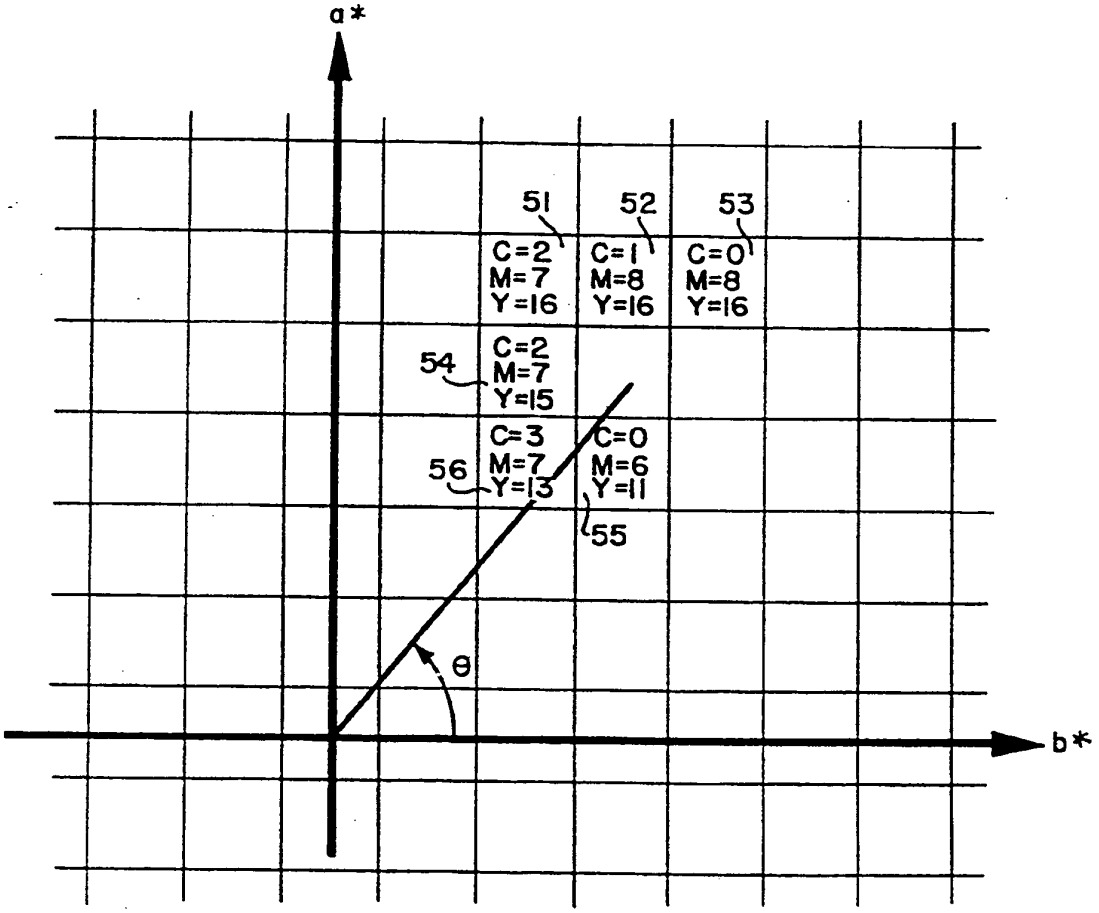
a*
51
52
53
C=2
M=7
Y=16
C=1
M=8
Y=16
C=0
M=8
Y=16
54
C=2
M=7
Y=15
C=3
M=7
Y=13
56
C=0
M=6
Y=11
55
θ
b*

FIG. 9

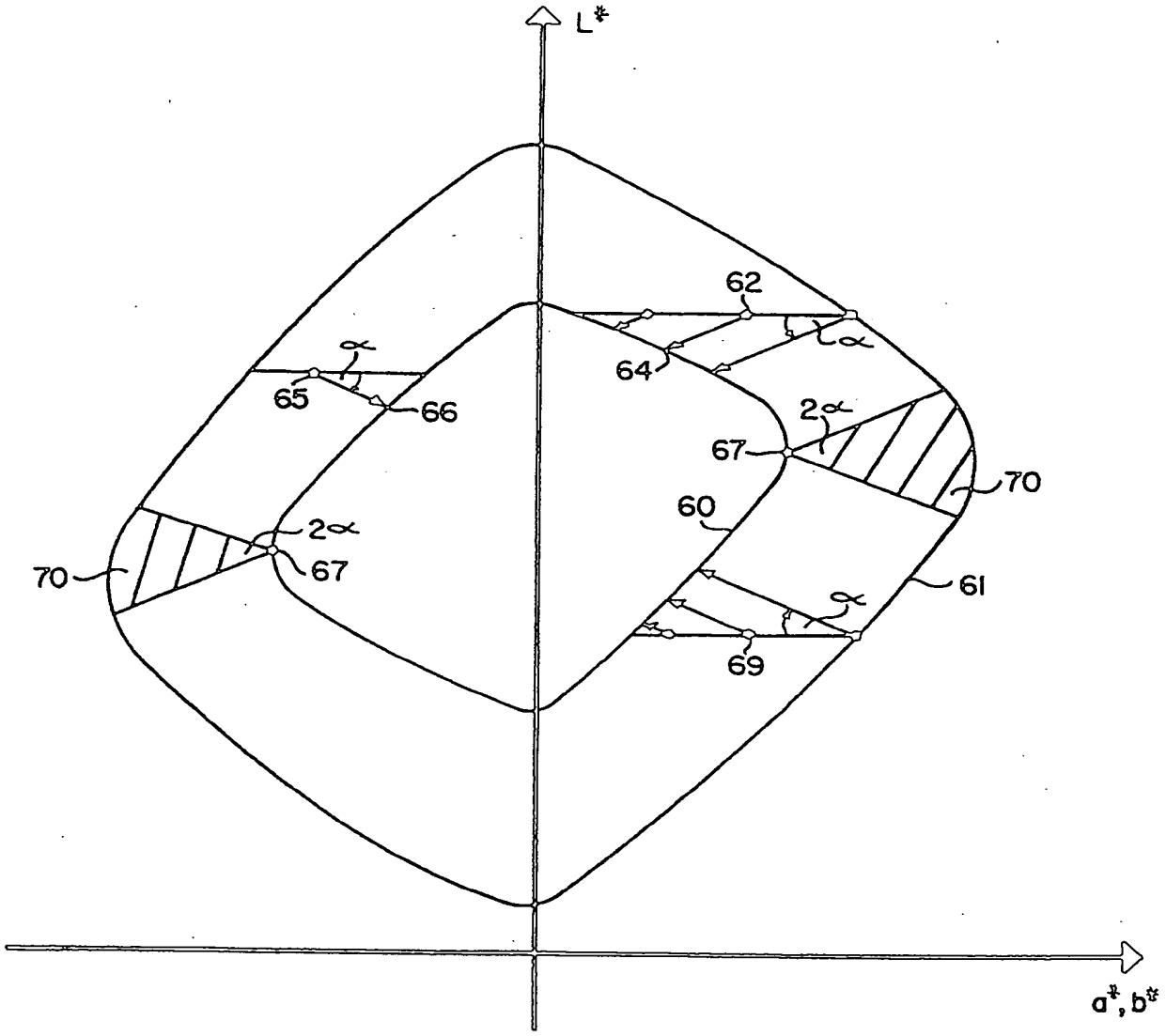

FIG. 10

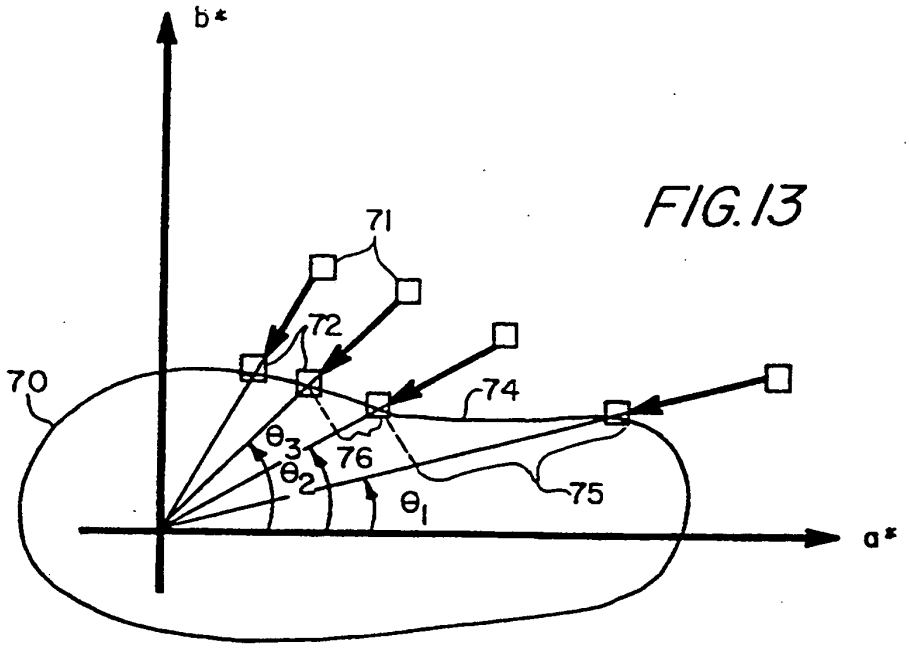

FIG. 13

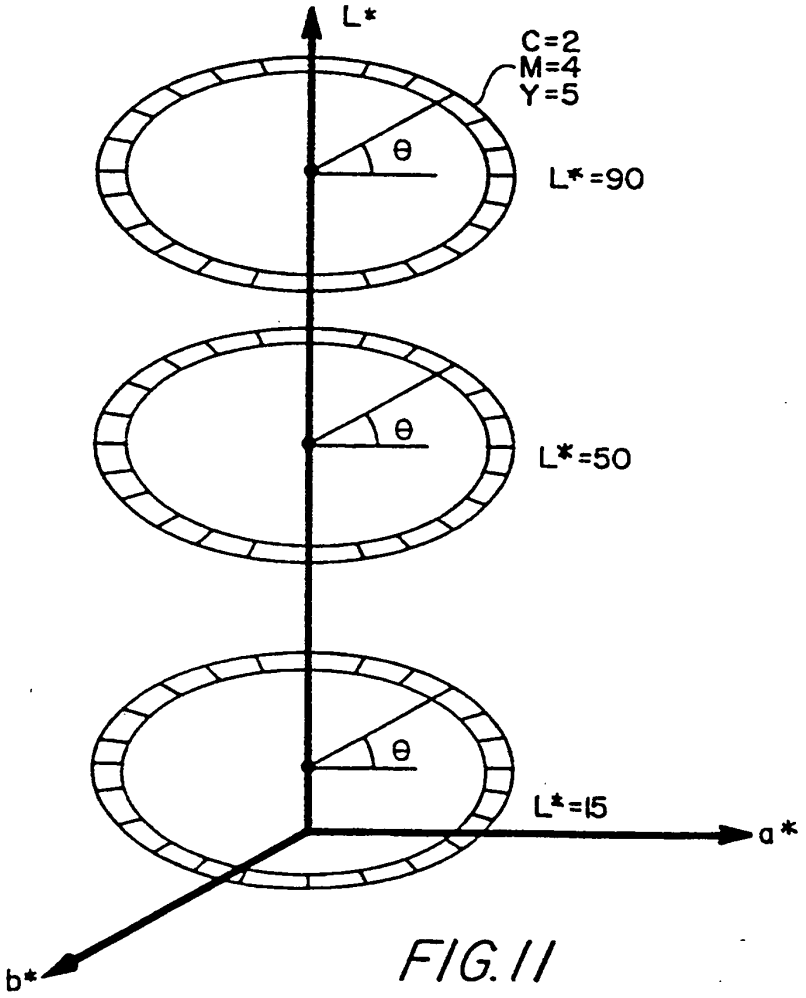

FIG. 11

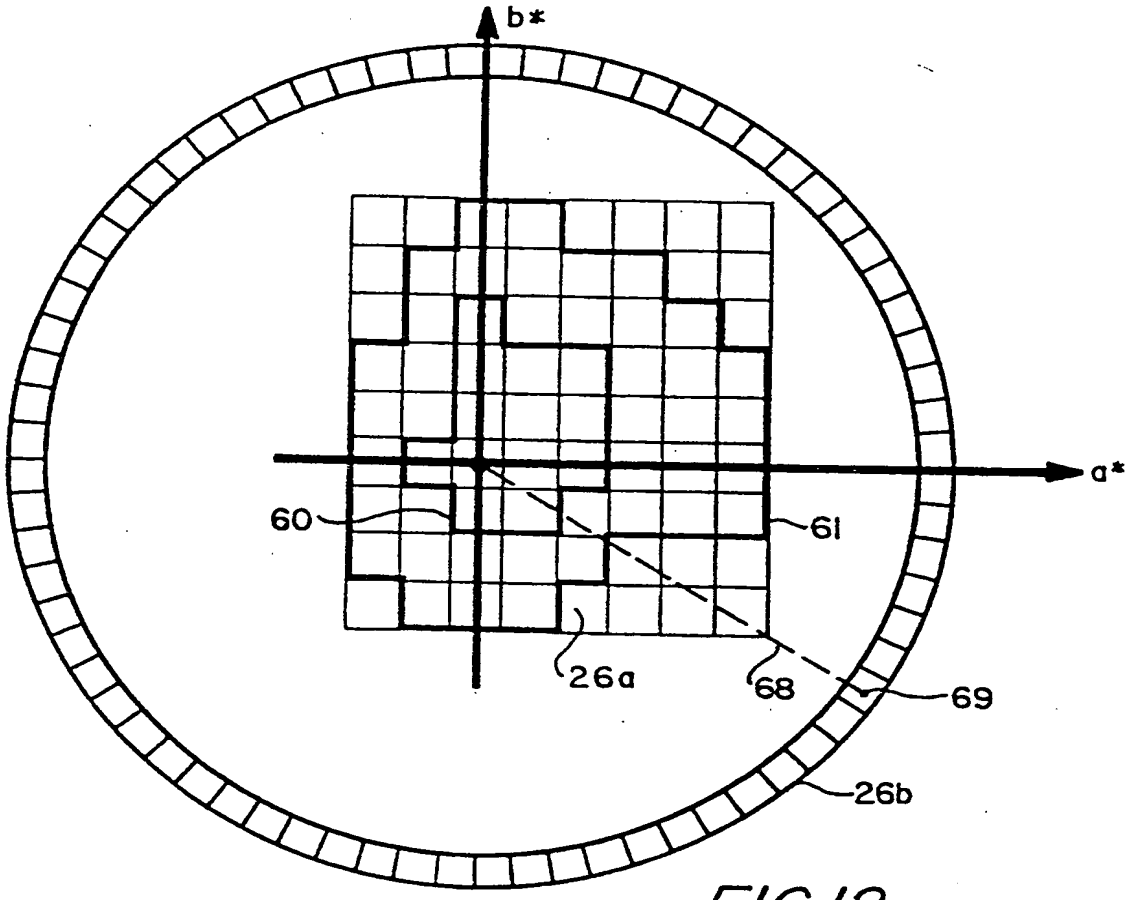

FIG.12

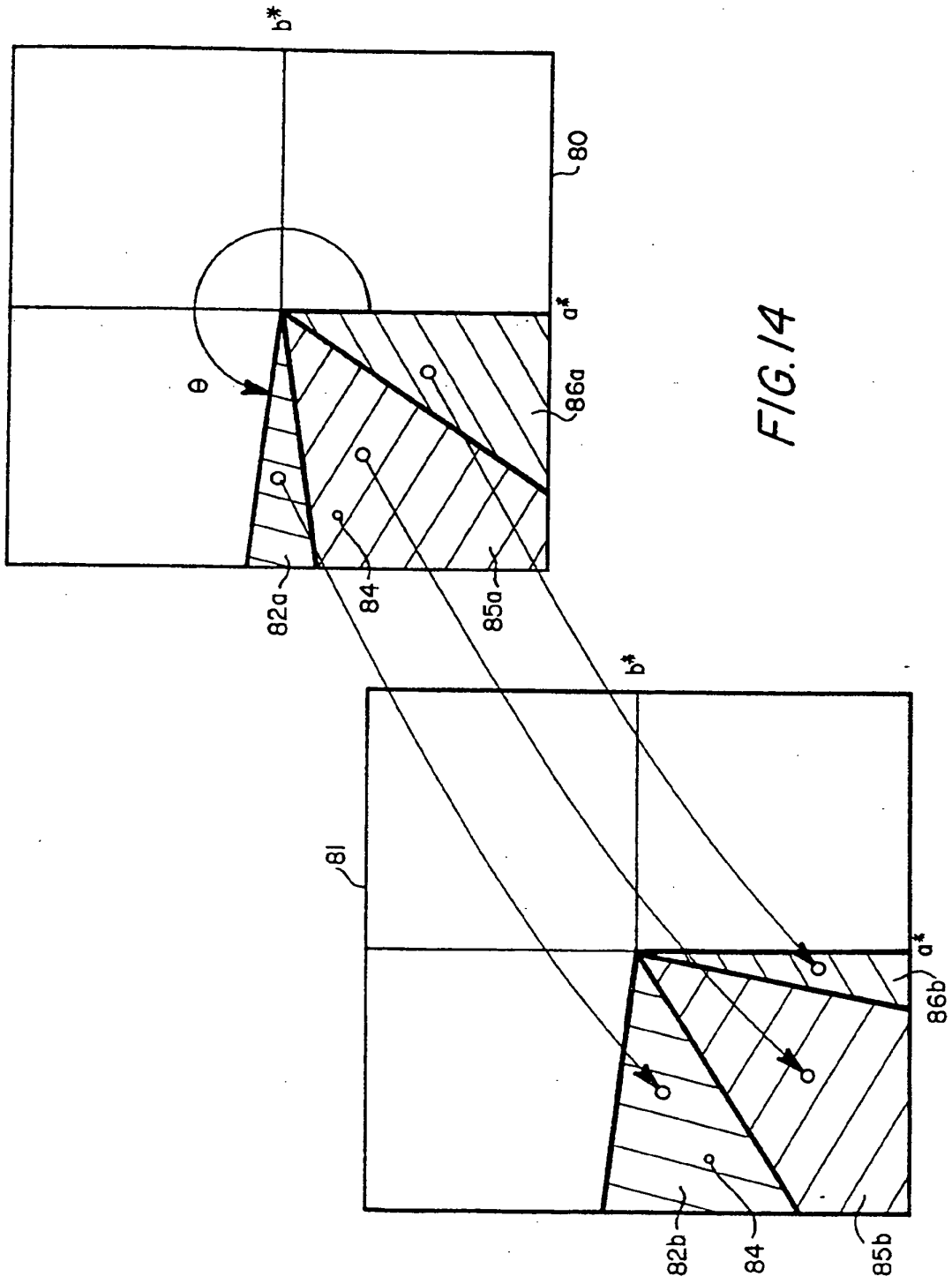

FIG. 14

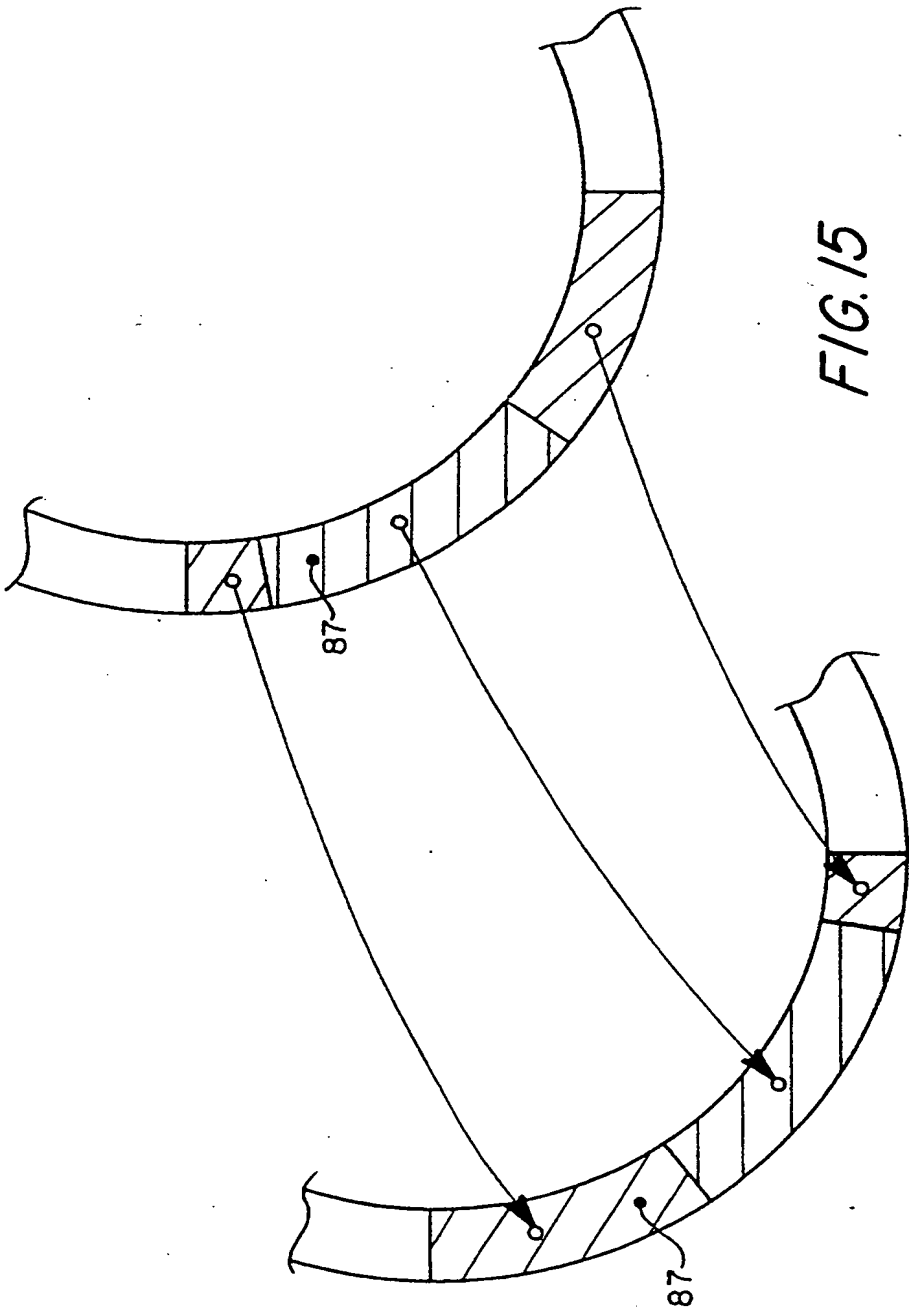

FIG. 15

(19) Europäisches Patentamt
Europ an Patent Offic
Office uropéen des brevets

(11) Publication number : **0 592 146 A3**

(12) EUROPEAN PATENT APPLICATION

(21) Application number : **93307757.0**

(22) Date of filing : **30.09.93**

(51) Int. Cl.[5] : **H04N 1/46**

(30) Priority : **28.10.92 US 967055**
05.10.92 US 956300

(43) Date of publication of application :
13.04.94 Bulletin 94/15

(84) Designated Contracting States :
DE FR GB IT

(88) Date of deferred publication of search report :
20.07.94 Bulletin 94/29

(71) Applicant : **CANON INFORMATION SYSTEMS, INC.**
3188 Pullman Street
Costa Mesa, CA 92626 (US)

(72) Inventor : **Ruetz, Brigitte**
255 Lassen drive
San Bruno, California 94066 (US)

(74) Representative : **Beresford, Keith Denis Lewis et al**
BERESFORD & Co.
2-5 Warwick Court
High Holborn
London WC1R 5DJ (GB)

(54) Color reproduction method and apparatus.

(57) Method and apparatus for color printing according to a printer table having high color smoothness for out-of-gamut colors. A color printer gamut edge is first determined and color primary values for colors within the printer gamut are calculated and inserted into the printer table. For transition colors, namely those colors outside the printer gamut edge but still within the printer table, color primary values are calculated by selecting a color from the printer gamut edge at a point which lies at a constant angle from the transition color in question. The same constant angle is used for each and every one of the transition colors. A border table is provided for colors outside of the printer table ; the color primary values for the border table are selected in the same manner as for the transition colors. According to the invention, it is possible to provide smooth color transitions for out-of-gamut colors, and in particular it is possible to provide monotonically increasing lightness in both the printer table and the border table and thereby avoid undesirable bands of darker colors within lighter colors.

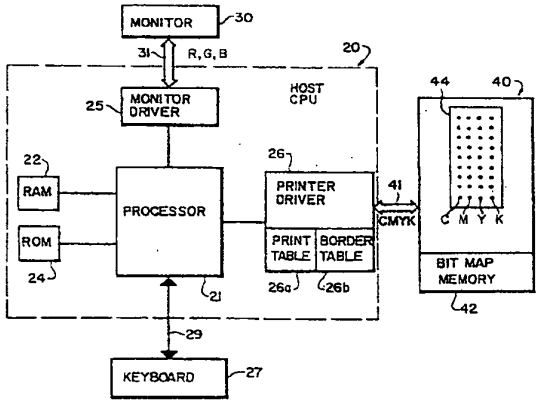

FIG. 3

EP 0 592 146 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 93 30 7757

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.5)
A	EP-A-0 321 983 (MINOLTA CAMERA KABUSHIKI KAISHA) 28 June 1989 * abstract; figures 5A,5B * * page 6, line 33 - page 7, line 46 *	1,7,15, 23-25	H04N1/46
A	EP-A-0 488 655 (KONICA CORPARATION) 3 June 1992 * abstract; figures 9-11 * * page 8, line 55 - page 9, line 56 *	1,7,15, 23-25	
A	GB-A-2 213 674 (XEROX CORPORATION) 16 August 1989 * abstract *	1,7,15, 23-25	

TECHNICAL FIELDS SEARCHED (Int.Cl.5)

H04N

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	16 May 1994	Revellio, H.S.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)